DOCUMENT RESUME

ED 190 049 IR 008 520

AUTHOR	Feurzeig, Wallace: And Others
TITLE	The LOGO-S Language and the Portable LOGO System. Volume I: Language and System Descriptions. Volume II: Program Listings. Report No. 3968.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	80
CONTRACT	SED-7611981
NOTE	580p.
EDRS PRICE	MF03/PC24 Plus Postage.
DESCRIPTORS	Computer Assisted Instruction: Computer Managed Instruction: *Computer Programs: *Computers: Educational Research: Man Machine Systems: Microcomputers: Minicomputers: *Programing Languages
IDENTIFIERS	*LOGO System

ABSTRACT

This two volume report presents descriptions of LOGO-S, an interactive programming language expressly designed for education, and the portable LOGO system, a computer software development. The modular structure of LOGO, along with its simple but powerful extension features, is naturally suited to problem solving in many problem domains with students at all levels of education. To make LOGO widely available on diverse computer systems the portable LOGO system is being developed. This software system greatly reduces the amount of work and the level of skill required to implement LOGO on minicomputers and microcomputers. The major products of this project are (1) the design and specification of a proposed standard LOGO language, LOGO-S: (2) the development of a portable LOGO system for implementation of LOGO-S: and (3) comprehensive documentation of LOGO-S and the portable LOGO system for use by implementors. Language description provides the syntax and a glossary of LOGO-S. Volume I presents an overview, including the implementation language, the host environment, global data structures of the LOGO machine, a description of the machine, and the LOGO-S environment; Volume II provides program listings for the current working version. Advice to implementors and a bibliography of 45 citations are included. (Author/RAA)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED190049

Report No. 3968

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

The LOGO-S LANGUAGE and The Portable LOGO System

VOLUME I: LANGUAGE AND SYSTEM Descriptions

Wallace Feurzeig
E. Paul Goldenberg
George Lukas
Vincent Manis
Richard Rubinstein
Robert Stachel

April 1980

The work described in this report was supported in part by the National Science Foundation under Contract No. SED-7611981. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Prepared by:

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

Prepared for:

The National Science Foundation

IR008530

MAY 1 1980

TABLE OF CONTENTS

Page

I PROJECT SUMMARY 7

1. INTRODUCTION 8

2. LOGO-S DEVELOPMENT 10

3. THE PORTABLE LOGO DESIGN 13

4. CONTINUED DEVELOPMENT OF LOGO 17

5. REFERENCES 19

II DESCRIPTION OF THE LOGO-S LANGUAGE 21

6. THE SYNTAX OF LOGO-S 22

7. THE GLOSSARY OF LOGO-S 28

7.1 Introduction 28
7.1.1 LOGO terminology 28
7.1.2 LOGO Primitives 32
7.1.3 Conventions used in this glossary 34
7.2 Predicates and Boolean Manipulation 35
7.2.1 Predicates that describe objects 35
7.2.2 Predicates that describe the workspace 37
7.2.3 Predicates that compare objects 38
7.2.4 Manipulating Boolean values 43
7.3 Object Manipulation 46
7.3.1 Manipulation of lists as tree structures 46
7.3.2 Manipulation of lists as unordered sets 48
7.3.3 Manipulation of lists and words as linear structures 50
7.3.4 Manipulation of certain properties of lists and words 53
7.4 Arithmetic 57
7.4.1 Types of numbers 57
7.4.2 Basic arithmetic operations 59
7.4.3 Trigonometric and Exponential operations 62
7.4.4 Random numbers 64
7.4.5 Formatting primitives 65
7.5 Procedures, Names and Arrays 70
7.5.1 Defining procedures 70
7.5.2 Editing procedures 73
7.5.3 Names 77
7.5.4 Arrays 78
7.6 The workspace 81
7.6.1 Contexts 81
7.6.2 Workspace Constituents 81
7.6.3 Workspace Manipulators 85
7.7 Control structure 88
7.7.1 Concepts of control structure in LOGO 88
7.7.2 Conditional Statements 88
7.7.3 Returning from procedures 90
7.7.4 Evaluators 92
7.8 Trapping, Error Handling and Debugging 94
7.8.1 Commands 97
7.8.2 Operations 99
7.8.3 Debugging 100
7.9 Input/Output 104
7.9.1 Transcription forms 104
7.9.2 Notes on the transcription of special characters 105
7.9.3 Terminal-oriented I/O 107
7.9.4 File-oriented I/O: Reading commands 110
7.9.5 File-oriented I/O: Writing commands 112

4

7.9.6 File-oriented I/O: General manipulation 114
7.9.7 File-oriented I/O: Properties 115
7.10 Properties 118
7.10.1 Primitives 118
7.10.2 Properties of Procedures, Names and Arrays 119
7.11 Turtles and other special devices 121
7.11.1 All turtles 121
7.11.2 Restricted turtles only 122
7.11.3 Turtle senses 125
7.11.4 Other devices 126
7.12 Miscellaneous 127
7.12.1 Top level control 127
7.12.2 Machine level primitives 128
7.13 Special Characters 130
7.13.1 Characters which interrupt execution 130
7.13.2 Editing Characters -- Deleting 130
7.13.3 Editing Characters -- Preserving/Retrieving/Viewing killed text 133
7.13.4 "White space" characters 134
7.13.5 Editing Characters -- Copy/Keeping input text 135
7.13.6 Characters without interrupt status 137
7.13.7 Delimiters 137
7.14 Error types 140
7.14.1 Errors occurring primarily during parsing 140
7.14.2 Errors occurring during evaluation 140

III A GUIDE TO IMPLEMENTATION 143

8. AN OVERVIEW OF THE PORTABLE LOGO SYSTEM 144

8.1 Components of Portable LOGO 144
8.2 General organization 145
8.3 Outline of Implementation Tasks 145

9. THE LOGO IMPLEMENTATION LANGUAGE 147

9.1 Introduction 147

9.2 The LIL machine 147
9.3 LIL Data Objects 149
    9.3.1 Vectors 149
    9.3.2 Numbers 150
    9.3.3 Strings 150
    9.3.4 Dynamically-created data objects 151
    9.3.5 Pseudo Heap Objects 152
9.4 Name tables 152
9.5 The LIL Instruction Set 153
    9.5.1 LIL Instruction Errors 153
    9.5.2 LIL Instruction Summary 154
    9.5.3 Basic Double Operand Instructions 156
    9.5.4 Basic Single Operand Instructions 157
    9.5.5 Double Length Arithmetic 158
    9.5.6 Control flow 158
    9.5.7 String operations 159
    9.5.8 Vector operations 161
    9.5.9 Vector operations for numbers only 161
    9.5.10 Conversion 162
    9.5.11 Symbol table manipulation 162
    9.5.12 Heap object prefix manipulation 164
    9.5.13 Stack operations 164
    9.5.14 Routine calling 165
    9.5.15 Flag operations 166
    9.5.16 Error Instructions 166
9.6 The LIL assembly language 166
    9.6.1 Basic structure 166
    9.6.2 Assembly expressions 167
    9.6.3 Address Expressions 167
9.7 LIL Assembly Directives (Pseudo-Operations) 168
    9.7.1 Preprocessing Directives 168
    9.7.2 Program Sectioning Directives 168
    9.7.3 Data Generation Directives 169
9.8 Coding conventions 170
    9.8.1 Register usage 170
    9.8.2 Conventions for LCGO-S Primitive Routines 171
    9.8.3 Error handling 171

10. THE HOST ENVIRONMENT 172

10.1 The Storage Allocator 172
10.2 Host interface routines 174
    10.2.1 Input/output 174
    10.2.2 Storage allocation 176
    10.2.3 Attention and clock interrupts 176

10.2.4 Program check handling 178
10.2.5 Timing 178
10.2.6 Floating Point Arithmetic 178
10.3 Initialization and Normal Termination 179
10.4 Storage Overflow 180
10.5 Abnormal termination 180

11. GLOBAL DATA STRUCTURES OF THE LOGO MACHINE 181

11.1 Words and Lists 181
11.2 Numbers 181
11.3 Procedure Directories 181
11.4 Internal Program Representation 182
11.4.1 Basic OM instructions 183
11.4.2 Control primitives 184
11.4.3 Infix operators 185
11.4.4 Examples 185
11.4.5 Summary of OM instructions 187
11.5 Name tables 188
11.6 The LOGO stack 188

12. THE LOGO MACHINE 190

12.1 Components of the LOGO machine 190
12.2 The parser 191
12.2.1 General organization 191
12.2.2 Lexical scanning 191
12.2.3 Building object code 191
12.2.4 Infix operators 191
12.2.5 Error handling 192
12.3 The evaluator 192
12.3.1 The operation of the evaluator 192
12.3.2 Error handling 193
12.3.3 RUN & VALUE 194
12.4 The Trap Handler 195
12.4.1 The Trap Tables 195
12.4.2 Receiving an interrupt 195
12.4.3 Dismissing an interrupt 196
12.4.4 TRAPRETURN 196
12.5 Initializing the LOGO Machine 196
12.5.1 Initializing the Name Tables 197
12.5.2 Bootstrapping LOGO-S 197

12.5.3 Creating a User Run Module 197

13. THE LOGO-S ENVIRONMENT 198

13.1 The monitor 198
13.2 The standard trapping procedures 199
13.2.1 The Error Reporting Subsystem 199
13.2.2 ERRORMESSAGE 199
13.2.3 BREAK (SUSPEND) 199
13.2.4 PANIC 199
13.3 The Line Editing Subsystem (EDITSTRING) 200
13.4 The Procedure Editing Subsystem 200
13.4.1 The Editing Variables 201

14. ADVICE TO IMPLEMENTORS 203

APPENDIX A. LOGO BIBLIOGRAPHY 205

INDEX 209

8

2

ABSTRACT

LOGO is a programming language expressly designed for education. During the past decade LOGO has been used in research and teaching at approximately forty universities. Nearly all this work has been done using LOGO implementations on the DEC PDP-10 computer system. The dissemination of LOGO-based educational research and the introduction of LOGO teaching into schools have been significantly limited by the lack of available LOGO implementations on computers other than the PDP-10. To make LOGO widely available on diverse computer systems, we have developed a portable LOGO system. This software system greatly reduces the amount of work and the level of skill required to implement LOGO on minicomputers and microcomputers, particularly those machines which will be substantially introduced into schools during the next decade. The major products of this project are:

1. the design and specification of a proposed standard LOGO language, LOGO-S,
2. the development of a portable LOGO system for implementation of LOGO-S, and
3. comprehensive documentation of LOGO-S and the portable LOGO system for use by implementors.

These materials have been requested by workers from several educational centers for use in LOGO implementations on a variety of computers.

FOREWORD

This project is designed to further the dissemination of LOGO-based education through making the LOGO programming language readily available on computer systems that are likely to be used in schools. To this end, together with representatives of the LOGO user community, we designed a LOGO language standard, LOGO-S. We then developed the portable LOGO system, a software system whose use greatly simplifies the task of realizing a LOGO-S implementation on a computer. The LOGO-S language specifications are given in this report, together with a comprehensive description of the portable LOGO system, written to provide a practical guide for implementors of LOGO-S.

Our work plan under this program comprised the following tasks.

1. Design and documentation of a LOGO language standard.
2. Development of a comprehensive strategy for LOGO implementation on a variety of computer systems.
3. Preparation of the preliminary version of a tutorial guidebook on LOGO implementation.
4. Implementation of LOGO on a minicomputer system and documentation of the programs.
5. Dissemination of the MIT LISPLOGO system to interested users.
6. Holding a working conference of the LOGO user community to present, and guide application of, the above work.
7. Consultation on additional implementation efforts of LOGO user communities.
8. Refinement and extension of tutorial guidebook incorporating results from user feedback.
9. Documentation of the LOGO guidebook and implementation programs.

These tasks have been completed. The associated research and the products of this work are described herein. Volume 1 contains detailed descriptions of the LOGO-S language and the portable LOGO system. Volume 2 provides listings of the portable LOGO system programs and of the first implementation produced using the portable LOGO system - the LOGO-S program for the DEC 20 minicomputer system.

In carrying out this work we organized two working conferences with members of the LOGO user community. The first LOGO users workshop was conducted on August 26-27, 1977 at the Massachusetts Institute of Technology. Several participants described their LOGO work in progress. (Work at the University of Edinburgh; MIT; University of Campinas, Brazil; the Open University, England; the University of Toronto; the University of British Columbia; McGill University and Darmstadt.) In the BBN presentations we suggested a strategy for LOGO dissemination and proposed our preliminary ideas for LOGO language extensions and for a standard LOGO language. Our proposals were extensively discussed, both at the meeting and through subsequent correspondence. Feedback from these discussions was productive, leading to improvements in ideas for the new LOGO language.

Participants at this workshop included:

- From BBN: Joe Berkovitz, Wallace Feurzeig, George Lukas, Victor Miller, Richard Rubinstein;
- From MIT: Hal Abelson, Howard Austin, Jean Bamberger, Andy diSessa, Paul Goldenberg, Ira Goldstein, Bob Lawler, Ron Lebel, Mark Miller, Seymour Papert, Dan Watt;
- From the University of Edinburgh: Jim Howe, Tim O'Shea, Sylvia Weir;
- From the University of British Columbia: Vincent Manis;
- From the University of Campinas: Fernando Curado;
- From Carnegie-Mellon University: Lee Gregg;
- From CNR, Rome, Italy: Giovanni Lariccia;
- From Darmstadt: Gerhard Fischer;
- From the University of Massachusetts, Amherst: Howard Peele;
- From McGill University: Guy Groen;
- From Open University, England: Marc Eisenstadt;
- From Stanford University: David Levine;
- From the University of Toronto: Neil Kaden, Bill Reeves;
- And from the U.S. Navy, NPRDC, San Diego: John Wolfe.

The second LOGO users workshop was conducted on May 18-19, 1979 at Bolt Beranek and Newman. At this workshop we presented the LOGO-S language specifications and described the portable LOGO system. Although participation was open to all LOGO users, the invitation stated that the discussions would primarily focus on implementation issues and would be relatively technical. The major agenda items were:

- Overview of the portable LOGO system
- Description of the LOGO Implementation Language (LIL)
- The portable LOGO subsystems
- Use of the Portable LOGO System
- LIL Implementation
- Host software interface requirements
- LOGO-S specifications.

The presentations were made by BBN LOGO Project staff. The DEC PDP-11 and TI 99/04 implementations of LOGO were also described by MIT programmers. Following the discussion of LOGO-S, a number of modifications were suggested mostly involving additional primitives or minor changes to the surface syntax. Some of these changes were subsequently incorporated.

The portable LOGO system and subsystems were described in considerable detail. Various design modifications, some aimed at trading storage requirements for gains in execution speed, were suggested. Some of these ideas generated improvements in the portable system design. Others were unacceptable since their use would have ruled out implementations on many of the small computers being acquired by schools.

Participants at the second workshop were:

- From BBN: Wallace Feurzeig, Paul Goldenberg, George Lukas, Vincent Manis, Charles R. Morgan, Richard Rubinstein, Robert Stachel;
- Boston University: Glenn Bresnahan;
- University of Edinburgh: Peter Ross;
- J. L. Rivers and Co.: Trenholm Turner;
- Lincoln-Sudbury Regional High School: Brian Harvey;
- McGill University: Guy Groen, Kiyoko Montpetit;

- MIT: Hal Abelson, Gary Drescher, Ron Lebel;
- University of Paris: Gerard Bossuet;
- University of Rhode Island: Carolyn Bartlett;
- University of Stuttgart: Gerhard Fischer;
- U.S. Social Security Administration: Ray Halverson;
- Xerox PARC: John Seely Brown.

We are grateful for the contributions made to LOGO-S and the portable LOGO system by those who participated in the two LOGO conferences. In addition we wish to acknowledge other valuable contributions made through correspondence and project visits by a number of individuals from the LOGO community, including: Ron Baecker, the University of Toronto; David Levine, Boston College; Gerhard Fischer, University of Stuttgart; Sterling Beckwith, York University; Neil Kaden, Northern Bell Canada; Neil Rowe, Stanford University; Ira Goldstein, Xerox PARC; Tim O'Shea, the Open University, England; Robert Donaghey, City University of New York; Charles R. Morgan, BBN; Robert Bobrow, BBN.

The authors were assisted in the task of LOGO design and implementation by many colleagues at BBN. We are especially grateful for the substantial contributions made by system programmers Victor Miller (now at IBM, Yorktown), Joseph Berkovitz, and Leonard Foner.

Finally, we would like to thank our NSF project officer, Dr. Andrew Molnar for his interest, guidance, and support throughout; for his personal assistance in resolving the several administrative issues that developed along the way; and for his long term support of LOGO educational research.

PART 1

PROJECT SUMMARY

1. INTRODUCTION

LOGO is an interactive programming language expressly designed for education. The modular structures of LOGO, along with its simple but powerful extensional features, are naturally suited to problem solving in many problem domains with students at all levels of education. Since its introduction in 1967, LOGO has been used in a variety of school and university courses, teaching experiments, and educational research projects. A bibliography of LOGO publications is included in Appendix A.

The goal of this project is to further the dissemination of LOGO- based educational activities by making LOGO more widely available throughout the educational community. Until now, LOGO has been implemented on very few computer systems and has only been readily available to users of a single machine, the DEC PDP-10. Some implementations made on other machines have used local non-standard operating systems; some have not been well developed or supported. No commercial versions of LOGO are yet available. Thus, most universities and schools have been unable to obtain LOGO versions to run on their computer systems. We hope to remedy this situation by the introduction of the portable LOGO system, a major product of this project. Through use of this software system, the task of implementing LOGO is made a great deal easier, well within the abilities, for example, of a computer science graduate student. The portable LOGO system was designed to help produce LOGO implementations on machines with distinctly different architectures, order complements, addressing systems, memory capacities, operating systems, and software resources. We expect the portable LOGO system to be used for producing LOGO implementations on a variety of computer systems during the next year or two. Among these are the microcomputer systems that will be most widely used in schools, as well as most minicomputer and main frame systems.

The object produced by the user of the portable LOGO system is a running LOGO language interpreter. Development of the portable LOGO system obviously requires that LOGO itself be carefully defined and specified. The LOGO language specification used as the basis for the portable LOGO system implementation will serve as a de facto language standard for a potentially very large user community. And unless present users feel favorably toward the specification, the portable system might not find a high level of initial acceptance. In defining the LOGO language to be used in the portable implementation we therefore actively solicited contributions from members of the LOGO community, taking into account user experience with previous LOGO designs as well as suggestions for extensions and changes.

The LOGO language specification that has been produced is called LOGO-S.

Since LOGO-S owes much of its design to contributions from a large segment of LOGO users, we feel justified in presenting it as a standard. LOGO-S is the second major product of this project.

Another major product is the implementation of LOGO-S on the DEC 20 computer system. This is the first application of the portable LOGO system. Our primary objectives in making this implementation were to test the portable LOGO system design concepts, to debug the portable system programs, and to provide an example for reference and for guidance to implementors.

The LOGO-S language design is the product of a ten-year evolutionary development. The background of LOGO language design and implementation efforts leading to the current LOGO-S specification is described in the next section.

2. LOGO-S DEVELOPMENT

The first LOGO interpreter was implemented at BBN in 1967 on a Scientific Data Systems SDS 940 computer [1]. This LOGO included procedures and recursion but had relatively few primitives and very limited editing, debugging and file manipulation facilities. It was tried out in preliminary school teaching in the summer of 1967.

This teaching experiment gave rise to a considerably richer LOGO language design which was implemented at BBN in 1968 on a DEC PDP-1 computer [2]. The PDP-1 LOGO was the first time-shared version. It was used in an extended series of LOGO-based teaching experiments, in elementary and junior high school mathematics classrooms, in 1969-1970.

In 1970/71 BBN implemented LOGO on the DEC PDP-10 under the TENEX operating system [3,4]. This is the version of LOGO which has been most widely disseminated. It looks very much like present LOGOs though it does not have general list structures, arrays, or floating point numbers.

These data types and other language extensions were provided on a PDP-10 implementation made at MIT in 1971 [5]. A larger and more elaborate LOGO system, called LISP LOGO, was developed at MIT in 1974 [6]. This version, which requires a PDP-10 with approximately 50K memory, is implemented in LISP and provides users with access to LISP functions. It is particularly suited for sophisticated computer science users and for applications such as artificial intelligence programming. MIT also developed a DEC PDP-11/40 LOGO under a local MIT operating system in 1974 [7]. This system has been since converted to run on the LSI-11. The MIT LISP LOGO and 11-LOGO designs incorporated a number of language extensions such as functions with unlimited arguments and error trapping facilities. MIT has recently implemented a more restricted LOGO on the Texas Instruments TI 99/04 [8].

Other interesting LOGO implementations have been made at Queen Mary College [9], at the University of British Columbia for the IBM 360/370 [10], and at Edinburgh a version called ELOGO [11] and a recent version for the LSI-11 based TERAK computer [12]. More restricted implementations have been made, for example, in BASIC on the Hewlett Packard 4000 and in CALL-360 on the IBM 360. We are aware of other implementations on ICL, Siemens, Danish NOVA, and other systems, but are not familiar with the details of these. We considered all known versions of LOGO in developing the LOGO-S specification. In addition, we considered constructs provided in other languages, particularly SMALLTALK, SIMULA, SNOBOL and PASCAL.

At the start of the project we polled LOGO users to find which language augmentations, relative to their current LOGO facilities, had the greatest importance for them. Substantial responses were made by researchers from Rutgers, Stanford, Edinburgh, Leeds, York, Xerox PARC, Darmstadt, Kiel, University of Western Ontario, and MIT. Users responded with interests in four main areas:

- New data types: inclusion of "real" arithmetic, arrays and list structures.
- File management: more flexible and general file structures permitting selective access and retrieval of data and programs.
- Program execution and control: error trapping and breakpoint facilities, parallel processing capabilities, extended editing functions.
- Graphics: a generalized graphics package for use with device-dependent interfaces.

We have taken these recommendations into account in the design of LOGO-S. The initial version of LOGO-S includes most of these requested augmentations.

At a meeting of LOGO users on August 26, 1977, following the International Joint Conference on Artificial Intelligence in Cambridge, Mass., we presented an initial proposal for LOGO treatment of a number of central issues related to names and naming, defining and editing procedures, control structures, arrays, lists, real numbers, graphics, debugging and error handling, and file operations [13]. Attendees included representatives from MIT, Darmstadt, Univ. of Edinburgh, National Research Council (Canada), Univ. of Toronto, CNR (Italy), York Univ., Boston College, Stanford Univ., University of Campinas (Brazil), Open University (England), Univ. of British Columbia, McGill University and BBN. Our proposals generated considerable discussion and suggestions for further changes, many of which have been adopted in LOGO-S.

The initial design of the LOGO-S language was substantially completed in early 1979. On May 18-20, 1979 we called a conference of interested LOGO users to report on the LOGO-S specification and the portable LOGO system design [14,15]. Attendees included representatives from Edinburgh, McGill, the University of Paris, Boston University, MIT, the University of Rhode Island, the U. S. Social Security Administration, the University of Stuttgart, Xerox PARC, General Turtle Inc., Lincoln- Sudbury Regional High School, and BBN.

The program included BBN presentations of the portable LOGO system design and subsystem components, and use of the system by implementors. MIT presentations described their PDP-11 and TI 99/04 implementations. The BBN presentation of the LOGO-S design met with general approval. Recommendations for modifications and extensions were proposed for subsequent consideration. The overall favorable response appears to assure acceptance by the larger user community. The present version of LOGO-S is described in Part II.

3. THE PORTABLE LOGO DESIGN

In designing LOGO-S our task was to integrate new ideas and forms with well-established ones with the object of creating a consistent, coherent and expressive language. Ostensibly, the task of designing the portable LOGO system afforded a great deal more scope for choice and development: no established ideas or materials had to be integrated into the system. Our major objective was that the scope of application of the system be very extensive - the target system is to be used for producing effective and efficient implementations for a large and varied set of machines and operating systems including microcomputers with relatively small internal storage. These considerations posed our most important constraints and guided our major design decisions.

In developing the portable LOGO design, we studied two existing portable implementation systems developed for other interactive high level languages: (1) a macro implementation of SPITBOL, an interactive version of SNOBOL4 [16], and (2) a macro implementation of INTERLISP [17]. The former is interesting in its documentation of the detailed policy decisions made to achieve portability. It is written in a low level PDP-11-like language called MINIMAL. The INTERLISP implementation has special value because LISP is so similar to LOGO in basic semantics and because the macros are carefully described.

Our portable implementation was designed to satisfy three key requirements.

Portability The implementation should produce effective programs across a variety of computers with different machine architectures, word lengths, instruction complements. Moreover, to generate LOGO systems suitable for the smaller computers widely used in schools, the implementation should yield fairly efficient programs. Storage requirements in particular must be kept moderate; the storage allocation programs must operate to accomplish this over a wide range of conditions.

Useability The user of our portable implementation will typically be a programmer with considerably less experience and competence than a professional systems programmer. The user's task of writing the machine-dependent programs necessary to make the portable implementation effective on his machine must therefore be clearly specified and relatively easy for him to accomplish.

Extensibility As the LOGO language is extended or otherwise modified, the task of making the corresponding modifications to the existing implementations must be made not only generally feasible but relatively easy. Thus, even though the user's initial implementation problem reduces to writing a well-specified "front-end" and interface, our portable implementation must be designed to anticipate subsequent needs for LOGO development. We must provide good facilities for changing substantial parts of the portable implementation program itself.

To satisfy these requirements, we have adopted a two-level implementation. LOGO itself is written in a high level language, LIL. LIL has been designed to facilitate the description and manipulation of the data structures (strings, lists, stacks, arrays, procedures) and the major programs (the primitive routines, parser, and evaluator) comprising the LOGO interpreter. The LIL program should be found highly readable and comprehensive by systems programmers. Subsequent changes in the LOGO-S language should, for the most part, be relatively easy to implement in LIL. The LIL processor can either be written in an assembly language or a high level language such as FORTRAN or PASCAL.

Some of the more complex primitives of LOGO-S are written as LOGO-S procedures instead of LIL subroutines. This can be a powerful and useful device. It provides a straightforward means for creating and testing new primitives as potential LOGO-S extensions.

Another design goal for the portable system was to produce relatively small size implementations. We want users to be able to fit medium-to-large student jobs with considerable amounts of recursion into a large microcomputer or small minicomputer configuration. Such a configuration is exemplified by an LSI-11 or a Z-80 based system with 56K bytes of memory and floppy disk as a secondary, file storage medium.

At the start of the project we made the major design decision, that space considerations outweigh execution time considerations, both for static and dynamic storage. This choice was made on the basis that lengthy time of execution, especially on stand-alone machines, is not a critical factor in determining user acceptance. Also, storage requirement on non-swapped systems seems to us to be a more absolute limitation on interpreter performance than is time. If one slows a computation down on an LSI-11 by a factor of 2, it will take twice as long to do, but if one doubles the space requirement of an interpreter on a small machine, one diminishes the size as well as complexity of the program that can be run at all. Also, some LOGO implementations are of interest on machines with less than 56K bytes.

A second major design decision was to develop an interpreter rather than a compiler. This choice was based on the call by value regime used by LOGO in parsing parameters, thus obviating the need for both source and object code to be kept in workspace. Also, interpreter output is much simpler to debug than compiler output. A compiler usually has the advantage of saving runtime storage because the compiler code itself can be discarded from core once the source code is compiled and execution is to begin. This is not possible in LOGO because new source code can be created during program execution.

A third important decision concerned the type of dynamic storage scheme to be utilized. Our LOGO storage environment has three major components. We have several name tables; we have code for the implementation itself; we have working storage consisting of one or more stacks and a "heap" in which various types of objects of varying length are stored. User-defined procedures are usually stored within the heap, possibly without the source code. The working storage is dynamically allocated during procedure definition and also as temporary or permanent structures are generated during execution. As structures are no longer referred to, the space they occupy may be reclaimed for other use by a space management program which is automatically invoked by LOGO as required.

Wegbreit gives a hierarchy of three dynamic storage allocation schemes in order of increasing power and complexity. In the first, the dynamically allocated storage contains no pointer at all. This is like the string space used in many BASIC implementations and is exceedingly easy to implement. Of course, only single level structures can be accommodated, a constraint we would have difficulty abiding by. The next level of complexity permits pointers in the dynamic storage with the restriction that any data element has exactly one pointer pointing to it, exactly one "owner." This restriction means that we can easily keep track of our free space as we execute, allocating and deallocating. Occasionally we will have to do a compaction when free storage becomes too fragmented but this is an easy operation if we include as part of each data element a backpointer to its parent. The highest level scheme is the general list structure which permits referential loops and thus general graph structures as well as the trees permitted in the second level scheme. The other significant advantage of such a multiple pointer system is that it does not require the copying of structures -- if we want a subtree we simply point to it instead of having to make a new copy as the "single owner" restriction requires.

For our initial LOGO implementation, we have chosen the "single owner" storage scheme, permitting generalized tree structures for great expressive power, while minimizing the amount of code and operating expense that would be required for a general garbage collection algorithm suited to more complex data types. Most LOGO applications are not hampered by the inability to represent general graphs directly. In general, this scheme permits lists to be represented more compactly, although this may not be true if the workspace contains data structures with many common substructures. This is a tradeoff against the loss in execution speed caused by the copying of data structures. Our choice for LOGO-S Version 0 reflects our concern for implementations on small machines. We are currently exploring a multiple-pointer implementation for later versions of LOGO-S.

The other choices made in specifying the portable system are documented in Part III. Assessment of the effects of these several decisions on the ease of developing a particular LOGO-S implementation and in the compactness and execution speed characteristics of the resulting program, will require operational experience with use of the system. An initial test has been made through the first application of portable LOGO in producing a LOGO-S on the DEC 20. This implementation appears to be efficient in both storage utilization and computational speed. It is documented in Volume II.

4. CONTINUED DEVELOPMENT OF LOGO

A detailed description of LOGO-S is included in Part 2 of this report. The LOGO-S specifications given there anticipate language extensions beyond those implemented in the current version, which is called version 0. Correspondingly, the program listings included in Volume 2 of this report are also for LOGO-S Version 0. These listings will be superseded in the near future as more extended versions of LOGO-S are produced.

We plan to continue development, both of LOGO-S and of the portable system. We plan to implement the extensions described in the LOGO-S glossary, which include new data types and new language primitives, as part of this continuing work. We will maintain future versions of LOGO-S, the corresponding language and system documents, and associated program listings. Thus, potential users of the portable LOGO system planning to implement LOGO-S are advised to write directly to BBN for current materials. Inquiries and requests concerning the most current editions of LOGO-S and the portable system should be directed to:

BBN LOGO Project
Department of Educational Technology and Instructional Systems
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, MA 02138

These continuing post-project developments extend the scope of the contract work and are not being supported by the National Science Foundation. They reflect the continuing interest of Bolt Beranek and Newman in LOGO.

We plan to produce further documentation of LOGO-S, including introductory and tutorial materials. It should be noted that the description of LOGO-S given here assumes a relatively informed reader already familiar with programming languages; it is not intended for beginning students of programming. The current documentation provides readers reference level materials, rather than introductory manuals. We expect most readers of the report to be already familiar with LOGO and interested in LOGO implementation using the portable system. Documentation for less informed readers will be forthcoming.

LOGO-S is a substantial language -- it comprises more than 200 primitives and special names. Yet, in some important respects (mostly relating to device-dependent facilities), it is incomplete. Users with character display terminals would like LOGO-S to provide two-dimensional display-oriented editing facilities. Users whose terminals have graphics processing capabilities would like a richer set of turtle drawing and animation facilities. We intend to provide specifications for some of these facilities at a later time. Thus, we expect LOGO-S to be a growing language. At the same time, we are concerned about users of small computers for whom even the present LOGO-S language might be too large. Specifically, we recognize that it might be necessary to define a subset of LOGO-S for users of microcomputer systems. We have not yet specified such a subset (MicroLOGO) because we are not sure that it will be necessary to do so. We are awaiting results of LOGO-S implementation efforts on LSI-11 and Z-80 based microcomputers to decide this issue.

We plan to serve as a clearing house to provide advice for users of the portable system and to distribute implementations of LOGO-S made with the portable system on various machines. We expect to make a few such implementations ourselves. We will try to make our implementations and those of other groups available to those interested in using LOGO. Finally, we plan to work with others in the LOGO community, in developing and publishing a variety of LOGO-based teaching materials, which are so greatly needed to further disseminate LOGO in education.

5. REFERENCES

1. The LOGO System, F. Frazier, et al, BBN Technical Memorandum, September 1967.
2. Programming Language as a Conceptual Framework for Teaching Mathematics, Final Report to National Science Foundation, (BBN Report No. 1889), W. Feurzeig, et al, November 1969.
3. Programming Languages as a Conceptual Framework for Teaching Mathematics, Vol. 4, The LOGO Processor. W. Feurzeig, et al, Final Report to National Science Foundation. NTIS No. PB-206919; ERIC ED057-582, June 1971.
4. LOGO Reference Manual, Addendum No. 1 and Addendum No. 2, BBN LOGO Notes, Oct.-Nov. 1971.
5. PDP-10 LOGO Description, W. Williams, MIT LOGO Project paper, 1971.
6. LLOGO: An Implementation of LOGO in LISP, I. Goldstein, et al. MIT LOGO Memo 11, June 1974.
7. A Glossary of PDP-11 LOGO Primitives, E. P. Goldenberg, LOGO Memo 16, March 1975.
8. A Glossary of PASCAL LOGO Primitives, MIT Note, April 1979.
9. LITL, LOGO-like Interactive Teaching Language, J. Hutchinson & M. Lamming, Computer System Laboratory, Queen Mary College, London, October 1976.
10. A Machine Independent Implementation of LOGO, V. S. Manis, Thesis, University of British Columbia, October 1973.
11. How to Work the LOGO Machine, B. duBoulay, T. O'Shea, Dept. of Artificial Intelligence, University of Edinburgh, November 1976.
12. Design of the TERAK LOGO System, P. M. Ross, Dept. of Artificial Intelligence, Univ. of Edinburgh, February 1979.
13. Notes on LOGO Language Specification for Portable Implementation, W. Feurzeig, G. Lukas, BBN, August 1977.
14. Notes on the BBN Portable LOGO Language, Educational Technology Department, BBN, May 1979.

15. The BBN Portable LOGO System, Preliminary Report, W. Feurzeig et al, BBN, May 1979.

16. MACRO SPITBOL - a SNOBOL4 Compiler, R. B. K. Dewar and A. P. McCann, Software-Practice and Experience, Vol. 7, 95-113, 1977.

17. The INTERLISP Virtual Machine Specification, J. Strother Moore II, Xerox Palo Alto Research Center, September 1976.

PART 2

DESCRIPTION OF THE LOGO-S LANGUAGE

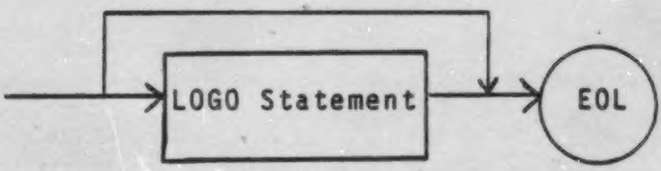
6. THE SYNTAX OF LOGO-S

This chapter describes the grammar of LOGO-S version 0 using syntax diagrams. (We expect that minor changes may be made in subsequent versions). All the more complex LOGO-S structures are shown. The following linear structures are not included:

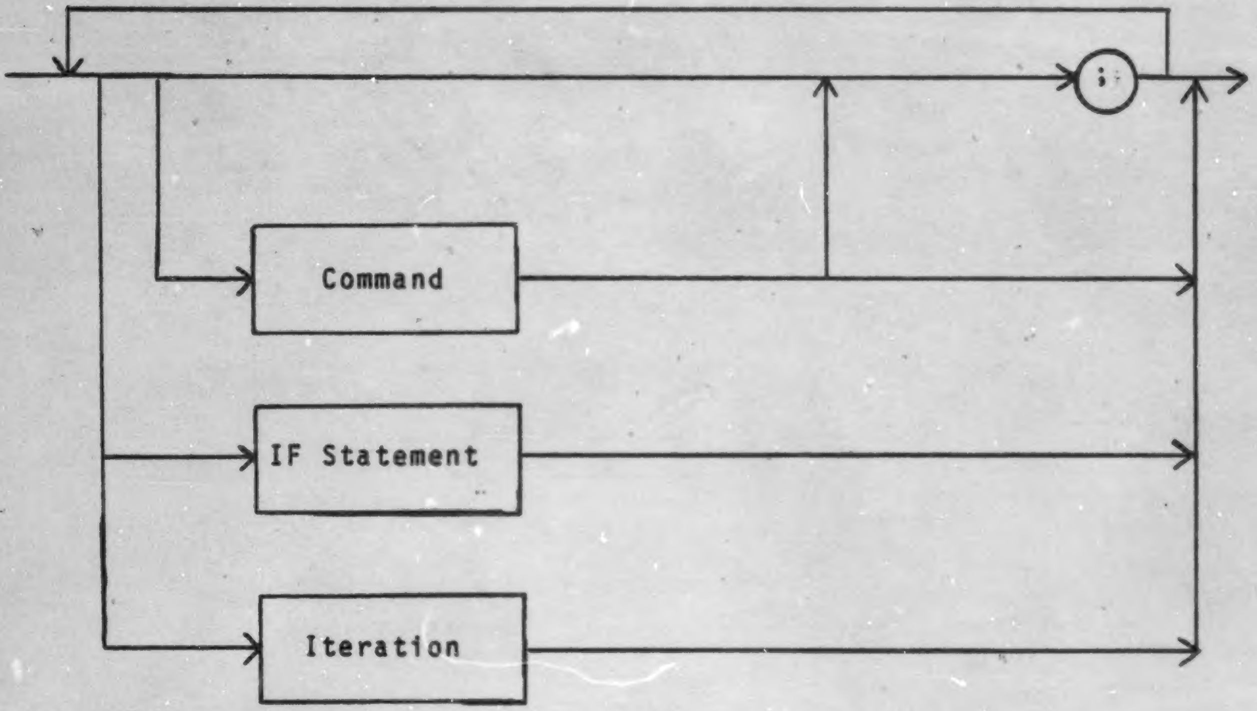
- **Word.** Any sequence of characters not including: left and right square brackets, parentheses, and braces; semicolons; double quotes; at signs (the superquote character); and control characters. (These characters may be included when prefixed by @).
- **Unsigned Integer.** Any sequence of decimal digits.
- **Name.** Any word not beginning with a decimal digit.
- **Procedure Name.** Any name which refers to a defined procedure.

Starting at the diagram labeled Command Line, the directed lines show all possible syntactically correct LOGO command lines; these are the basic executable units of LOGO-S. In the diagrams, non-terminal symbols for LOGO-S forms are shown in rectangular boxes. The terminals are enclosed in circles. EOL is an implicit terminal which represents the end of a command line. Other terminals are the literal symbols used.

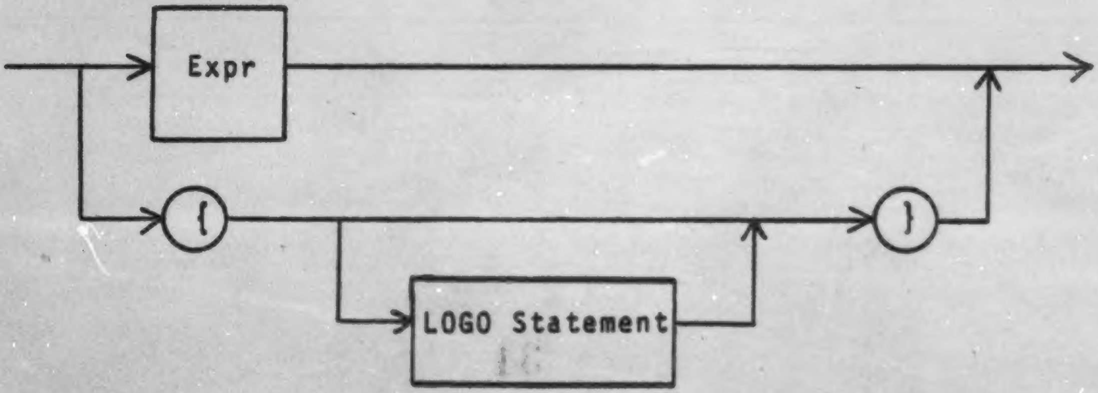
Command Line



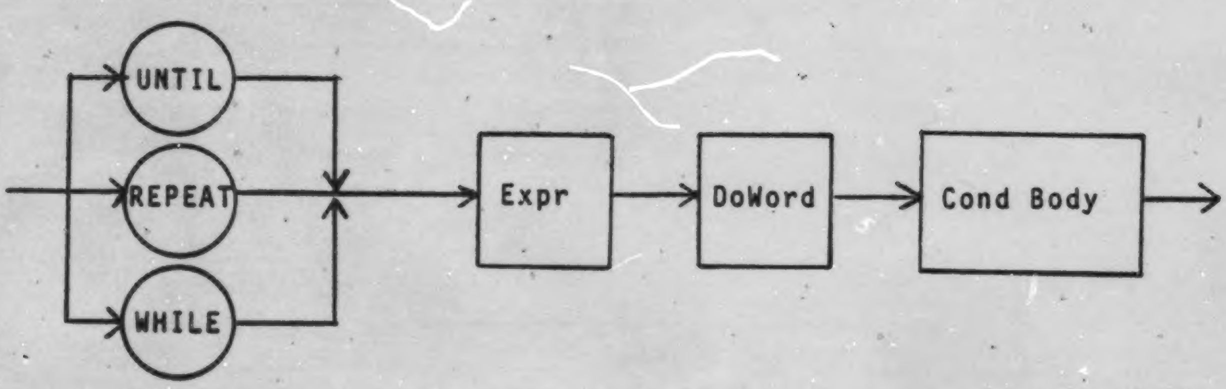
LOGO Statement



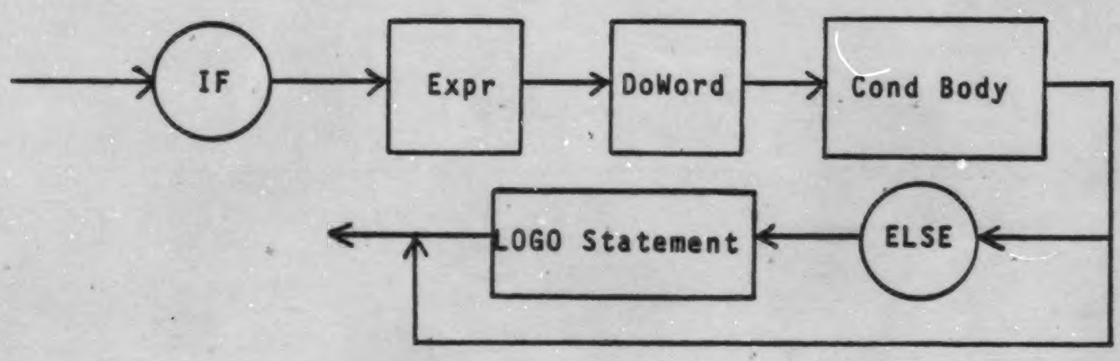
Command



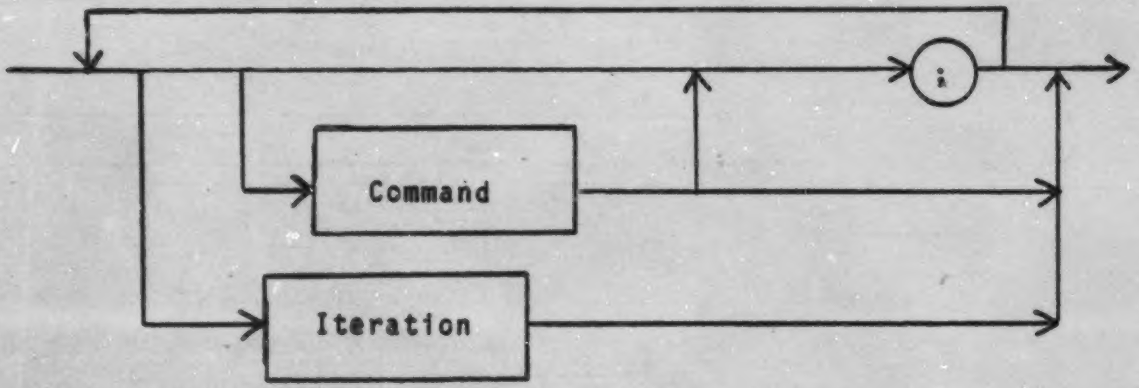
Iteration



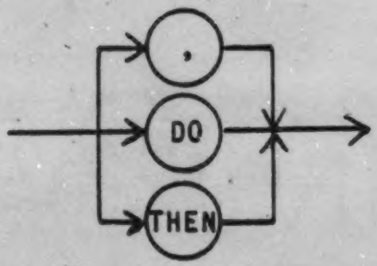
IF Statement



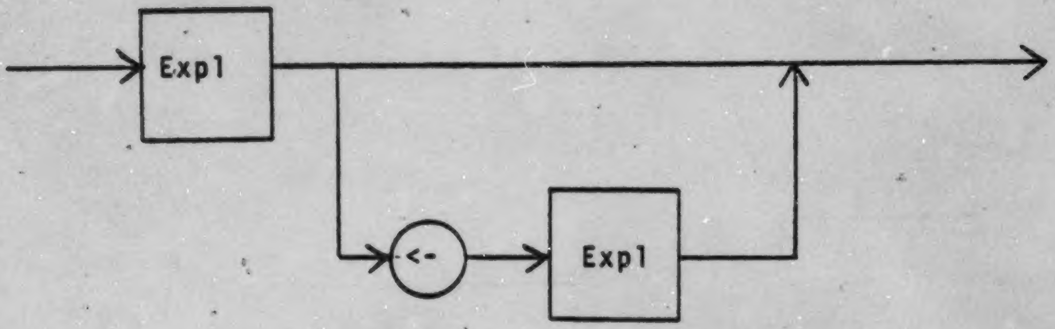
Cond Body



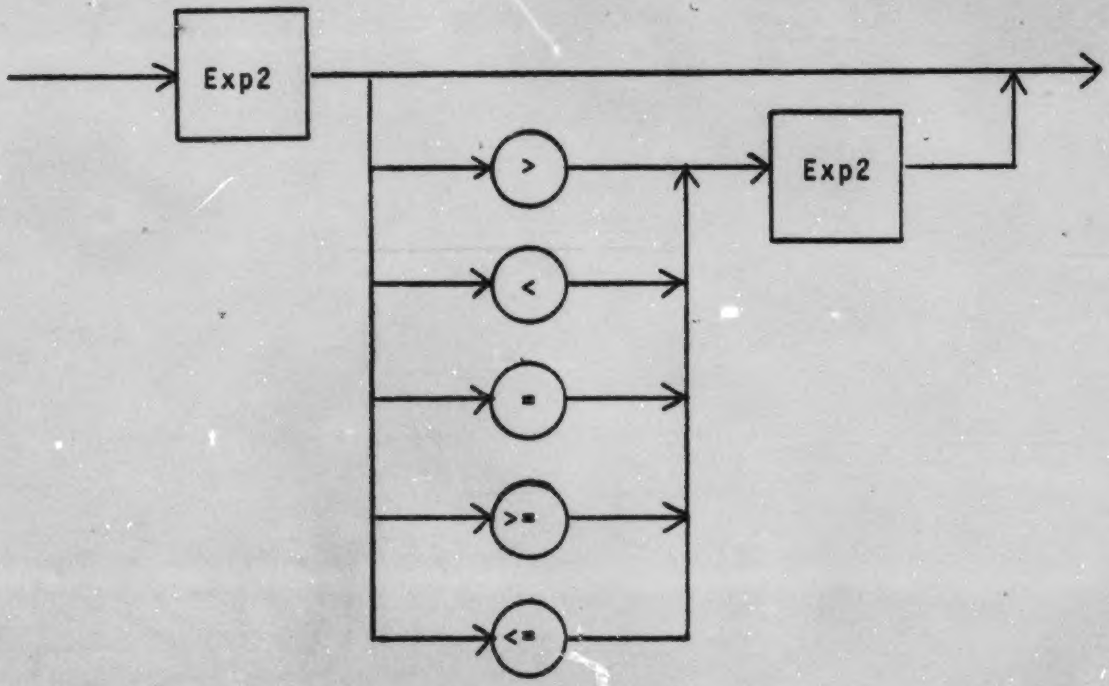
DoWord



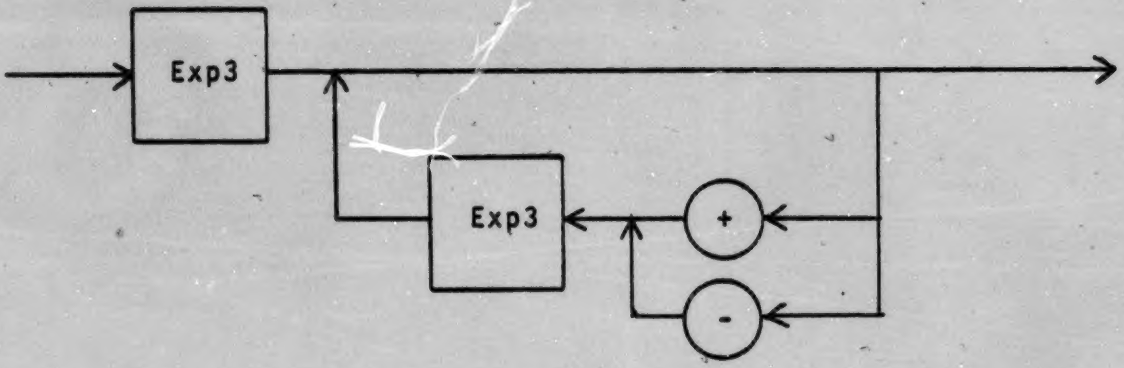
Expr



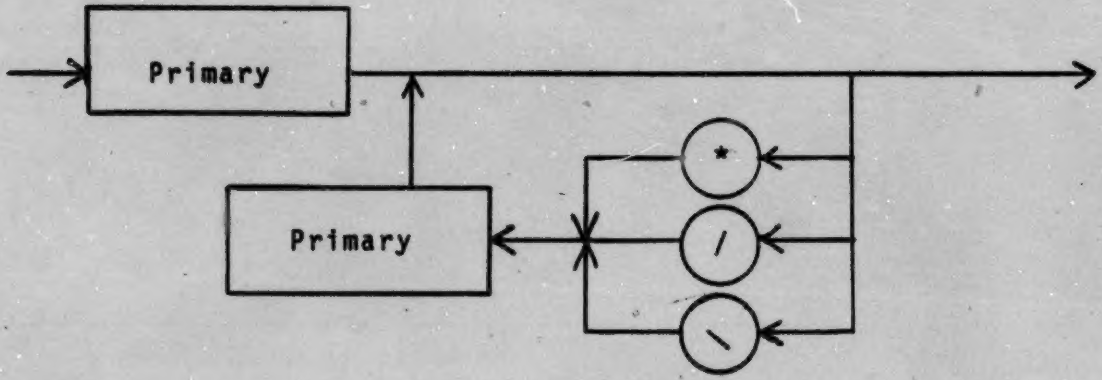
Exp1



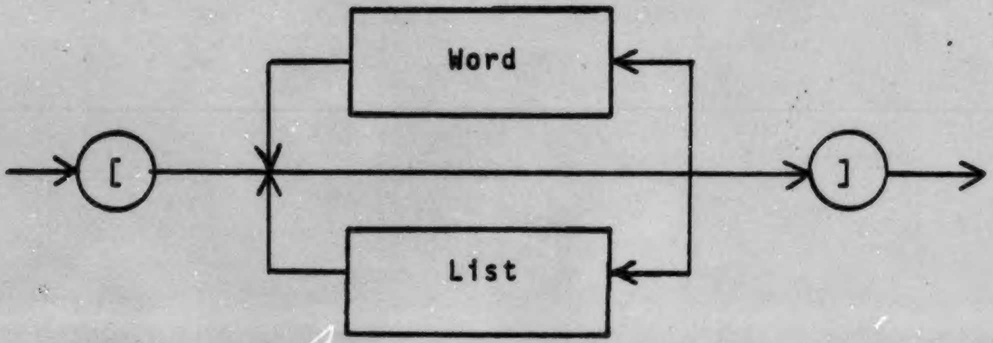
Exp2



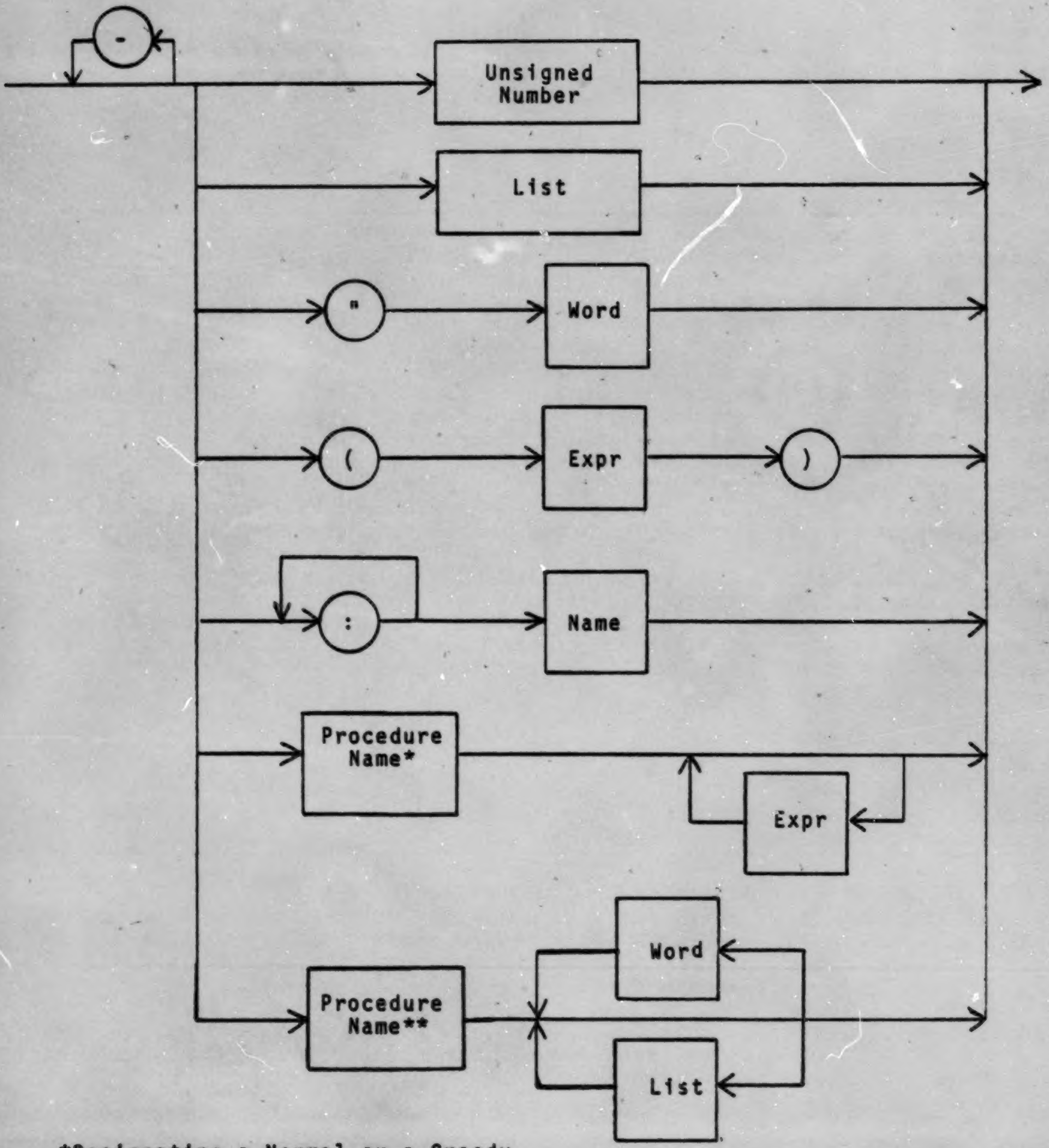
Exp3



List



Primary



*Designating a Normal or a Greedy Procedure (see Glossary)
**Designating a Quoting Procedure (see Glossary)

7. THE GLOSSARY OF LOGO-S

7.1 Introduction

This is the Glossary document for BBN LOGO-S, a new portable version of the LOGO language developed at Bolt Beranek and Newman. LOGO-S is written in an implementation language called LIL (for LOGO Implementation Language), permitting it to be brought up quickly on a variety of machines of varying architecture and size. For details of the work necessary to make LOGO-S work on your machine, see the LOGO Implementer's Guide.

LOGO programs are created by writing procedures. A procedure may contain one or more lines, and each line a LOGO command (or possibly more than one). A large number of pre-defined procedures are provided as part of the LOGO system, for use in creating user procedures. These built-in procedures are called primitives. They are described in this document, along with a number of other predefined components of the LOGO environment, specifically system names and control characters.

This section of the glossary describes LOGO-S terminology, the overall organization of the sections on primitives, and the conventions used in glossing them. The next eleven sections describe the primitives, themselves. The two final sections describe the special characters used in LOGO-S, and the classification of error types.

7.1.1 LOGO terminology

A variety of terms are used in this document with specific meanings in terms of the LOGO-S system. This alphabetical listing, a "glossary's glossary," summarizes these terms.

7.1.1.1 Primitives, procedures and programs

Abbreviations: Short spellings which expand (optionally) in type-in to full spellings of procedure names. LOGO-S version 0 does not contain abbreviations, but treats all abbreviated names as synonyms.

Boolean values: Values True or False. In LOGO-S, version 0, the case-independent text strings "True and "False serve as Boolean values.

Greedy procedures:	Procedures that do not specify in advance the precise number of arguments that they require, and, on invocation, accept a variable number of inputs (including none). Scope of inputs is limited syntactically by the end of the command, by semi-colon, or by parentheses surrounding the procedure name and all inputs to it. Arguments to greedy procedures are evaluated before the first line of the procedure is executed and are passed to the procedure as a single list. By contrast, see Normal and Quoting procedures.
Infix form:	To accommodate common algebraic notation, certain primitives have an infix form: a symbol representing the procedure but placed between the (two) inputs. An example is A + B which is equivalent to Sum A B.
Normal procedures:	Procedures that specify in advance the precise number of arguments (including none) that they require. Arguments to normal procedures are evaluated before the first line of the procedure is executed. By contrast, see Greedy and Quoting procedures.
Predicate:	These procedures answer a True-or-False question about LOGO structures by examining or comparing them and returning the values True or False. LOGO-S convention generally appends a question mark to the end of predicate names.
Prefix form:	LOGO procedures are invoked with the name of procedure preceding the inputs to the procedure (and separated by spaces), i.e., with the procedure before the inputs. For contrast, see Infix form.
Primitives:	Primitives are the commands and operations provided with the LOGO-S system, sometimes referred to as the "built-ins." These are the indivisible elements of user-defined procedures. User-defined procedures may be given the properties "Primitive, "NoText and "NoChange (see **Properties** in section 7.1.1.4) to make them indivisible like the built-in primitives.
Procedures:	Procedures are the building blocks of LOGO programs. They are of two types -- primitive (built-in) procedures and non-primitive (user-defined) procedures. A user-defined procedure is composed of other procedures

and may include recursive calls to its own definition. A procedure has a name (used to invoke it), and may have input variables and local variables. Its body consists of numbered command lines which specify normal sequencing and execution.

Quoting procedures:	Quoting procedures differ from normal procedures only in that they do not evaluate their arguments automatically, before the body of the procedure is entered. Depending on the procedure itself, and perhaps also on certain conditions (as in If statements), inputs may be evaluated by the procedure itself.
Synonyms:	Alternate names for procedures. Unlike abbreviations, synonyms are maintained as typed (i.e., are not expanded to other spellings).

7.1.1.2 Statement Types

Command:	A command is a procedure that does not return a value (i.e., does not have an output). Normally, commands are at the left-most side of statements.
Control delimiters:	These are syntactic markers (sometimes keywords) that may appear in statements to separate components of the statement. They include Then, Do, the comma (,), and Else. In addition to these, which are used only within conditional statements, the semi-colon (;) may be used anywhere to separate two statements that appear in one logical line of LOGO code.
Control word:	Certain LOGO syntactical constructions have special effects with respect to one or more statements within their scope. These include If, Repeat, While, and Until.
Operation:	An operation is a procedure that returns a value (i.e., has an output). It may not be used at the left-most side of a statement, and contrasts in this way with commands. See command.
Statement:	The basic unit of LOGO syntax, consisting of a command and its inputs (which may be complex).

7.1.1.3 Data Types

Array:	A named and indexed aggregate of data objects. Arrays are not data objects themselves.
Flat list:	A flat list is a data structure containing an ordered sequence of words.
List:	A list is a data structure containing an ordered sequence of words or lists.
Name:	A word serving as the symbolic name for a data object.
Number:	A LOGO word with a numerical value. LOGO-S provides integer, fixed point, and floating point numbers. (Further specifications are provided in the section on arithmetic.)
Object:	A LOGO word (including a number) or list. No other workspace constituent (e.g., procedure or array) is referred to as an object.
Thing:	The data object associated with a name is known as the thing of that name.
Word:	Words are the smallest units of data in LOGO. A word may have any number (including 0, see empty object) of characters in it. All characters may be used in words, but typing conventions require that some characters, such as <space>, must be inserted in special ways if they are to be interpreted as data rather than as separators between commands and data.

7.1.1.4 The LOGO Programming Environment

Contexts:	Tree-like structures, each node of which is a workspace. See further documentation on contexts, section 7.6.1. Contexts have not been implemented in LOGO-S version 0.
Descriptors:	Refer to different classes of workspace constituents. Workspace constituents include all items creatable by the user, e.g., procedures, names, and arrays.
Workspace:	The user's programming "space" containing all user created structures (procedures, names, arrays).

7.1.2 LOGO Primitives

1. **Predicates and Boolean Manipulation.** These procedures answer a True-or-False question about LOGO structures by examining or comparing them and returning the values True or False. Predicates provide the input to conditional statements (see "Control Structure" below), ultimately determining the computational course a procedure follows. This section describes the general purpose predicates that determine the type of LOGO structures, the relationship among structures, and the state of the workspace. Also included are the Boolean manipulators that combine the results of predicate operations. Special purpose predicates, e.g., those pertaining to graphics or workspace properties, are not described in this section, but rather in those sections devoted to the related special topic.

2. **Object Manipulation.** These procedures provide the basic LOGO capabilities for creating, modifying, and analyzing words and lists. Included are primitives for adding and removing elements of lists, accessing particular elements of lists, and "transcribing" lists into printable form. Also included are primitives that make convenient the use of lists to represent trees and unordered sets.

3. **Arithmetic.** This collection of basic functions provides arithmetic capabilities for handling numbers. In LOGO, there are three kinds of numbers: integers of arbitrarily great size, fixed point numbers with a defined number of decimal places of precision, and floating point numbers with fixed number of significant digits (precision) and a magnitude expressed as a power of ten. The usual arithmetic functions are provided for numbers: addition, subtraction, multiplication, division, remainder, negative, and square-root. Procedures to generate random numbers, to convert between the number types, and to manipulate the precision of fixed and floating numbers are also provided. Trigonometric and logarithmic functions will be provided in later releases, but are documented within the glossary.

4. **Procedures, Names and Arrays.** These are the primitives that create and access the definitions of all workspace constituents. They create and edit procedures, name and access data objects (i.e., create and read variables), and define and manipulate arrays as aggregates of data objects.

5. **The workspace.** The LOGO workspace is the user's fundamental programming environment. Primitives described in this section provide structured lists of the various workspace constituents

and provide the means for showing their definitions or removing them from the workspace.

6. Control Structure. LOGO-S provides a variety of control structures to enhance its expressive capability. Conditional statements such as If, While, Until, and Repeat allow conditional execution and repetition of a command. Other commands regulate the returning of control and values from one procedure to the procedure that called it and the evaluation of LOGO expressions that are stored as data.

7. Trapping, Error Handling, and Debugging. Facilities are provided for controlling a program's behavior during asynchronous events including, but not restricted to, the occurrence of errors during execution. The user is able to write procedures that perform special actions when these conditions occur, and the debugging package permits examination of the state of execution of the user's program in the context of an error or trap.

8. Input/Output. Primitives are provided for reading and writing information to or from the user's terminal and other devices such as disk or character-oriented special purpose devices that may be available on the host computer.

9. Properties. Workspace constituents may be tagged with a variety of properties indicating the kind of access to that structure that a user is permitted to have. Variables, for example, may be declared Read-Only. Various structures (names, procedures and arrays) may be declared "invisible" under normal circumstances.

10. Turtles and other special devices. LOGO is frequently used with specialized peripheral devices for graphics, robotics, music and other non-print functions. This section briefly sketches a preliminary set of control primitives for floor and screen turtles. Extended graphics capabilities and control of other devices will be developed for later versions of LOGO-S.

11. Miscellaneous. General interaction with LOGO's monitor and extensions beyond LOGO's internal code are discussed here. A primitive is provided to allow the accessing of non-LOGO programs from within LOGO.

Many sections are introduced by prefatory remarks further clarifying the primitives and the structures and concepts they manipulate.

7.1.3 Conventions used in this glossary

The description of each primitive, reserved system name, keyword or special character begins in a separately numbered subsection. The title line of the subsection lists first the primitive (or character, etc.), and then any abbreviations, synonyms and infix forms. If the primitive is a command (i.e., it does not return a value), the word "Command" is included at the end of that title line. The second line of the description generally indicates the Syntax of the primitive in the form:

Primitive Input1 Input2 etc.

If the primitive takes no inputs, this line is generally omitted and the second line contains the entry:

Input: None.

The input line otherwise indicates the acceptable inputs to that primitive. The output line indicates the type of the object returned by the primitive. If the primitive is a command (i.e., if it has already been indicated that the primitive does not return any value), there is no output line.

The behavior of the primitive is described in the semantics paragraph. Immediately following the semantics paragraph, evaluation errors are indicated. It should be noted that these are NOT the error messages given to the user, but only indicate the kinds of errors that can occur during the evaluation of the primitive. It should also be noted that only the evaluation errors, not parse errors, are indicated. Thus, incorrect number of inputs is not included as an error type. Finally, when the primitive takes no inputs, no errors are indicated.

Infix forms, abbreviations, synonyms and greedy forms (if any) are noted next. Examples, LOGO definitions (i.e., semantically equivalent LOGO procedures) and miscellaneous notes may follow where deemed necessary for clarification. It should be pointed out that the LOGO definitions, where provided, are for clarification of the meaning or behavior of the primitive and are not generally the full definitions of the primitive. For example, these LOGO definitions do not include any error-checking.

7.2 Predicates and Boolean Manipulation

7.2.1 Predicates that describe objects

7.2.1.1 Empty? E?

Syntax: Empty? Object
Inputs: Any object
Output: A Boolean value.
Semantics: Returns True if input is the empty object, False otherwise.
Errors: None.
Synonym: E?

7.2.1.2 Word? W?

Syntax: Word? Object
Inputs: Any object
Output: A Boolean value.
Semantics: Returns True if Object is a LOGO word, False otherwise.
Errors: None.
Synonym: W?

7.2.1.3 List? L?

Syntax: List? Object
Inputs: Any object
Output: A Boolean value.
Semantics: Returns True if Object is a LOGO list, False otherwise.
Errors: None.
Synonym: L?

7.2.1.4 FlatList? Flat?

Syntax: FlatList? Object
Inputs: Any object
Output: A Boolean value.
Semantics: Returns True if Object is a "flat" list (i.e., a list containing words only), False otherwise.
Errors: None.
Synonym: Flat?

7.2.1.5 Coordinates?

Syntax: Coordinates? Coordinates Tree
Inputs: A flat list of integers and an arbitrary list
Output: A Boolean value.
Semantics: Returns True if node described by Coordinates exists, False otherwise.
Errors: List argument in wrong format.
List expected, word found.
Note: Not in LOGO-S version 0.

7.2.1.6 Number? N?

Syntax: Number? Object
Inputs: Any object.
Output: A Boolean value.
Semantics: Returns True if the input object is a number, False otherwise.
Errors: None.
Synonym: N?

7.2.1.7 Integer? Int?

Syntax: Integer? Object
Inputs: Any object.
Output: A Boolean value.
Semantics: Returns True if the input object is an integer, False otherwise.
Errors: None.
Synonym: Int?

7.2.1.8 Zero? Z?

Syntax:	Integer? Object
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if the input object has the numerical value 0, False otherwise.
Errors:	None.
Synonym:	Z?

7.2.2 Predicates that describe the workspace

7.2.2.1 Array?

Syntax:	Array? Name
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if Name names a defined array, False otherwise.
Errors:	None.
Note:	Not in LOGO-S version 0.

7.2.2.2 SaveFile?

Syntax:	SaveFile? Name
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if Name names an existing save file, False otherwise. (See Save and Get.
Errors:	None.

7.2.2.3 Thing?

Syntax:	Thing? Name
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if Name designates a defined name (i.e., has a Thing that is not Undefined), False otherwise.
Errors:	None.

7.2.2.4 Procedure?

Syntax:	Procedure? Name
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if Name designates a procedure (including both user defined procedures and ones marked with the property "Primitive), False otherwise.
Errors:	None.

7.2.2.5 Primitive?

Syntax:	Primitive? Name
Inputs:	Any object.
Output:	A Boolean value.
Semantics:	Returns True if Name names a primitive or a procedure that has the property "Primitive, False otherwise.
Errors:	None.

7.2.3 Predicates that compare objects

7.2.3.1 Equivalent? Eq? =

Syntax: Equivalent? Arg1 Arg2

Inputs: Two objects.

Output: A Boolean value.

Semantics: If both inputs are numbers and are numerically equal, Eq? returns True. If either input is not a number, True is returned if inputs are object equivalent (that is, they have the same type, the same text disregarding alphabetic case, and the same list structure if they are lists). In all other cases, False is returned. For case sensitive comparisons, use Identical?.

Example:

```
>Print Eq?  7  07 ; Print (7 = 07)
True
True
>Print ( [diff [ALPHA] case] = [DIFF [alpha] CASE] )
True
>Print ("Yes = "yes)
True
>Print ( [Diff [List] Nest] = [Diff [List Nest]] )
False
```

Errors: None.

Infix form: =

Synonym: Eq?

7.2.3.2 Identical? Id?

Syntax: Identical? Object1 Object2
Inputs: Two objects.
Output: A Boolean value.
Semantics: Returns True if the two inputs are object identical (i.e., of the same type, composed of the same characters in the same order, of the same alphabetic case, and of the same list structure if they are lists), False otherwise. Thus, the strings "04 and "4 are not object identical, although they are numerically equivalent, and the strings "Cat and "cat are not object identical, although they are textually equivalent. For case insensitive comparisons, use Equivalent?, or Raise objects before comparing.
Errors: None.
Synonym: Id?

7.2.3.3 Greater? >

Syntax: Greater? Arg1 Arg2
Inputs: Two words.
Output: A Boolean value.
Semantics: If both inputs are numbers, Greater? returns True if Arg1 is strictly numerically greater than Arg2. If either input is not a number, returns True if Arg1 follows Arg2 in strict alphabetical order (disregarding alphabetic case). In all other cases, False is returned. For case sensitive comparisons, use Follows?.
Errors: Word expected, list found.
Infix form: >
Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.3.4 GreaterEq? GE? >=

Syntax: GreaterEq? Arg1 Arg2
Inputs: Two words.
Output: A Boolean value.
Semantics: If both inputs are numbers, GreaterEq? returns True if Arg1 is strictly numerically greater than or equal to Arg2. If either input is not a number, returns True if Arg1 follows (in strict alphabetical order) or is object equivalent to Arg2 (disregarding alphabetic case). In all other cases, False is returned. For case sensitive object comparisons, use Not Precedes?.
Errors: Word expected, list found.
Synonym: GE?
Infix form: >=
Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.3.5 Precedes?

Syntax: Precedes? Object1 Object2
Inputs: Two words.
Output: A Boolean value.
Semantics: Returns True if Object1 precedes Object2 in strict alphabetical order (upper case preceding lower case), otherwise False. For case insensitive comparisons, use Less?, or Raise objects before comparing.
Errors: Word expected, list found.
Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.3.6 Less? <

Syntax: Less? Arg1 Arg2
Inputs: Two words.
Output: A Boolean value.
Semantics: If both inputs are numbers, Less? returns True if Arg1 is strictly numerically less than Arg2. If either input is not a number, returns True if Arg1 precedes Arg2 in strict alphabetical order (disregarding alphabetic case). In all other cases, False is returned. For case sensitive comparisons, use Precedes?.
Errors: Word expected, list found.
Infix form: <
Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.3.7 LessEq? LE? <=

Syntax: LessEq? Arg1 Arg2
Inputs: Two words.
Output: A Boolean value.
Semantics: If both inputs are numbers, LessEq? returns True if Arg1 is strictly numerically less than or equal to Arg2. If either input is not a number, returns True if Arg1 precedes (in strict alphabetical order) or is object equivalent to Arg2 (disregarding alphabetic case). In all other cases, False is returned. For case sensitive comparisons, use Not Follows?.
Errors: Word expected, list found.
Synonym: LE?
Infix form: <=
Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.3.8 Follows?

Syntax: Follows? Object1 Object2

Inputs: Two words.

Output: A Boolean value.

Semantics: Returns True if Object1 follows Object2 in strict alphabetical order (upper case preceding lower case), otherwise False. For case insensitive comparisons, use Greater?, or Raise objects before comparing.

Errors: Word expected, list found.

Note: The order assigned to non-alphabetic characters depends on the collation order of the machine on which this primitive is implemented.

7.2.4 Manipulating Boolean values

7.2.4.1 True On

Inputs: None.

Output: Literal string True.

Semantics: Returns the literal string, True. This string (regardless of alphabetic case) is recognized as the Boolean value True by primitives which take Boolean values as input.

Synonym: On

Note: Not in LOGO-S version 0.
Present implementation uses "True for the Boolean value.

7.2.4.2 False Off

Inputs: None.

Output: Literal string False.

Semantics: Returns the literal string, False. This string (regardless of alphabetic case) is recognized as the Boolean value False by primitives which take Boolean values as input.

Synonym: Off

Note: Not in LOGO-S version 0.
Present implementation uses "False for the Boolean value.

7.2.4.3 Not

Syntax: Not Boolean

Inputs: A Boolean value.

Output: A Boolean value.

Semantics: Returns False if its input is True, True if its input is False.

Errors: Boolean expected, non-Boolean found.

7.2.4.4 Both

Syntax: Both Bool1 Bool2

Inputs: Two Boolean values.

Output: A Boolean value.

Semantics: Returns True if both *Bool1* and *Bool2* are True.

Errors: Boolean expected, non-Boolean found.

Greedy form: All

Note: Both inputs are evaluated, even if the first argument evaluates to False.

7.2.4.5 Either

Syntax: Either Bool1 Bool2

Inputs: Two Boolean values.

Output: A Boolean value.

Semantics: Returns True if either or both of *Bool1* and *Bool2* are True.

Errors: Boolean expected, non-Boolean found.

Greedy form: Any

Note: Both inputs are evaluated, even if the first argument evaluates to True.

7.2.4.6 All

Syntax: All Bool1 Bool2 ... BoolN

Inputs: Any number of Boolean values.

Output: A Boolean value.

Semantics: Evaluates all inputs (left to right) and returns True if all inputs evaluate to True.

Errors: Boolean expected, non-Boolean found.

Note: All inputs are evaluated, even if a False is discovered before completing the argument list.

7.2.4.7 Any

Syntax: Any Bool1 Bool2 ... BoolN

Inputs: Any number of Boolean values.

Output: A Boolean value.

Semantics: Evaluates all inputs (left to right) and returns False if all inputs evaluate to False (i.e., True if any evaluate True).

Errors: Boolean expected, non-Boolean found.

Note: All inputs are evaluated, even if a True is discovered before completing the argument list.

7.3 Object Manipulation

LOGO-S, version 0, provides two kinds of data structures: words and lists. Words are arbitrary length strings of characters. Lists are ordered sequences of LOGO objects. LOGO Objects are either words or lists.

Lists may be conceptualized (and used) as representations for

- o unordered sets (each top-level element of the list representing an element of the set);
- o tree-structures (each non-terminal node of which is represented by a sub-list and terminal branch of which is a word);
- o unidimensional (or multidimensional, compact) arrays of LOGO objects.

Primitives have been designed to provide expressive facilities to handle LOGO lists with these three representations.

LOGO objects (words or lists) may be assigned as the value of a "name" (LOGO variable), may be passed as arguments to procedures and may be stored as elements of an array. Other LOGO structures, the array and the procedure, cannot be handled in the same way, although their names can be passed as words. (The text of a procedure can also be passed as a list, and in future versions of LOGO-S, conversion between arrays and lists may be implemented.) Throughout this glossary, the word "object" refers only to the data types "word" or "list."

7.3.1 Manipulation of lists as tree structures

7.3.1.1 Coordinates C

Syntax: Coordinates Object Tree
Inputs: An arbitrary object and a list or
A single character and a word.
Output: List of list(s) of integers
Semantics: Returns coordinates of the node(s) at which Object is found in Tree. Returns :Empty if Object is not found.
Errors: Single character word expected, something else found.
(Tree is word and Object is not single character.)
Synonym: C
Note: Not in LOGO-S version 0.

7.3.1.2 Node N

Syntax: Node Coordinates Tree

Inputs: A flat list of integers (or a single integer) and an arbitrary object

Output: An object

Semantics: Returns object found at specified node in Tree. If Tree is a word, returns the single character found at specified element.

Errors: Argument out of range.
(tree is a word and coordinates is not an integer.)
Number expected, non-numeric object found.

Synonym: N

Note: Not in LOGO-S version 0.

7.3.1.3 ButNode BN

Syntax: ButNode Coordinates Tree

Inputs: A flat list of integers (or a single integer) and an arbitrary object

Output: An object of same type as Tree

Semantics: Returns a list (or word) consisting of Tree with node Coordinates pruned out. If Tree is a word, a single character at node Coordinates is removed.

Errors: Number expected, non-numeric object found.

Synonym: BN

Note: Not in LOGO-S version 0.

7.3.1.4 NodePut NP

Syntax: NodePut Object Coordinates Tree
Inputs: An arbitrary object, a flat list of integers (or a single integer) and an arbitrary list or
A word, a integer and a word.
Output: An object of same type as Tree
Semantics: Inserts Object in Tree at node Coordinates (before, and not replacing, previous object there, if any) and returns the new tree object.
Errors: Argument out of range.
List argument in wrong format.
Number expected, non-numeric object found.
List expected, word found.
Synonym: NP
Note: Not in LOGO-S version 0.

7.3.1.5 ReplaceNode RN

Syntax: ReplaceNode Object Coordinates Tree
Inputs: An arbitrary object, a flat list of integers (or a single integer) and an arbitrary list or
A word, a integer and a word.
Output: An object of same type as Tree
Semantics: Returns a list (or word) consisting of Tree with original contents of node Coordinate replaced by Object.
Errors: Argument out of range.
List argument in wrong format.
Number expected, non-numeric object found.
List expected, word found.
Synonym: RN
Note: Not in LOGO-S version 0.

7.3.2 Manipulation of lists as unordered sets

7.3.2.1 Union
Syntax: Union SetA SetB
Inputs: Two lists
Output: A list
Semantics: Returns the list designating the union of the input sets.
Errors: List expected, word found.
Note: Not in LOGO-S version 0.

7.3.2.2 Intersection
Syntax: Intersection SetA SetB
Inputs: Two lists
Output: A list
Semantics: Returns the list designating the intersection of the input sets.
Errors: List expected, word found.
Note: Not in LOGO-S version 0.

7.3.2.3 SetDifference
Syntax: SetDifference SetA SetB
Inputs: Two lists
Output: A list
Semantics: Returns the list of all elements of SetA which are not elements of SetB.
Errors: List expected, word found.
Note: Not in LOGO-S version 0.

7.3.2.4 Compact
Syntax: Compact Set
Inputs: A list
Output: A list
Semantics: Returns a subset of Set in which no elements are duplicated (i.e., all elements are different).
Errors: List expected, word found.
Note: Not in LOGO-S version 0.

7.3.3 Manipulation of lists and words as linear structures

7.3.3.1 First F

Syntax: First Object
Inputs: Any object
Output: Object
Semantics: If input object is a list, returns the first element of the list, which may be a word or list itself. If the input is a word, returns the first character of the word.
Errors: None.
Synonym: F

7.3.3.2 ButFirst BF

Syntax: ButFirst Object
Inputs: Any object
Output: Object
Semantics: If input is a list, returns the Object with its first element pruned off. If the input is a word, returns the Object with its first character pruned off. Note that the returned object is always of the same type as the input object.
Errors: None.
Synonym: BF

7.3.3.3 Last L

Syntax: Last Object
Inputs: Any object
Output: Object
Semantics: If input is a list, returns the last element of the list, which may be a word or list itself. If the input is a word, returns the last character of the word.
Errors: None.
Synonym: L

7.3.3.4 ButLast BL

Syntax: ButLast Object
Inputs: Any object
Output: Object
Semantics: If input is a list, returns the Object with its last element pruned off. If the input is a word, returns the Object with its last character pruned off. Note that the returned object is always of the same type as the input object.
Errors: None.
Synonym: BL

7.3.3.5 FirstPut Fput

Syntax: FirstPut Element List
Inputs: Any object and a list.
Output: A list.
Semantics: Returns a list whose First is Element and whose ButFirst is List.
Errors: List expected, word found.
Synonym: Fput

7.3.3.6 LastPut Lput

Syntax: LastPut Element List
Inputs: Any object and a list.
Output: A list.
Semantics: Returns a list whose Last is Element and whose ButLast is List.
Errors: List expected, word found.
Synonym: Lput

7.3.3.7 Word W

Syntax: Word Word1 Word2
Inputs: Two words
Output: A word
Semantics: Returns a single word composed of the concatenation of the two input words attached in order with no separator.
Errors: Word expected, list found.
Synonym: W
Greedy form: LongWord
Greedy Abbrev: LW

7.3.3.8 Sentence S

Syntax: Sentence Object1 Object2
Inputs: Two objects.
Output: A list
Semantics: If both input objects are lists, Sentence returns a list whose elements are the elements of the two input lists. If either input object is a word, Sentence first converts that input to a single element list and then proceeds as above. Compare with List and FirstPut.
Errors: None.
Synonym: S
Greedy form: LongSentence
Greedy Abbrev: LS

7.3.3.9 List

Syntax: List Object1 Object2
Inputs: Two objects.
Output: A list
Semantics: Returns a list whose elements are the two input objects (not the elements of the objects, as with Sentence). Compare with Sentence and FirstPut.
Errors: None.
Greedy form: LongList
Greedy Abbrev: LList

7.3.3.10 SubObject

Syntax:	SubObject LowLim HighLim Object
Inputs:	Two positive integers and an arbitrary object.
Output:	An object of same type (word or list) as Object.
Semantics:	Returns object (of same type as Object) with elements from LowLim through HighLim (inclusive).
Errors:	Number expected, non-numeric object found. Argument out of range. (Negative or non-integer)

7.3.4 Manipulation of certain properties of lists and words

7.3.4.1 Count

Syntax:	Count Object
Inputs:	Any object
Output:	Non-negative integer
Semantics:	If the input is a list, count returns the number of (top level) elements of the list. If the input is a word, count returns the number of characters in the word (actual data, not transcribed form). Count of the empty object is 0.
Errors:	None.

7.3.4.2 Member M

Syntax:	Member Element Object
Inputs:	An arbitrary object and a list or A single character and a word.
Output:	A non-negative integer.
Semantics:	Returns position of Element within Object (with 1 for first position). Returns 0 if Element is not a top-level element of Object. Error results if Object is a word and Element is not a single character.
Errors:	Single character word expected, something else found.
Synonym:	M

7.3.4.3 ABC

	ASCII
Syntax:	ABC Character
Inputs:	Single character word
Output:	Non-negative integer
Semantics:	Returns the numerical character code for the input character. (On ASCII systems, returns the ASCII value of the character in decimal.)
Errors:	Word expected, list found. Single character word expected, something else found.
Synonym:	ASCII
Note:	On non-ASCII systems, the synonym ASCII may not be provided.

7.3.4.4 Char

Syntax:	Char CharacterCode
Inputs:	Non-negative integer
Output:	Word
Semantics:	Returns the character corresponding to the input character code.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.3.4.5 Raise

Syntax:	Raise Object
Inputs:	Any object
Output:	Object
Semantics:	Returns an object of the same type as the input (word or list) in which all alphabetic characters are transcribed in upper case and all other characters are returned unaltered.
Errors:	None.
Note:	Not in LOGO-S version 0.

7.3.4.6 Lower

Syntax: Lower Object
Inputs: Any object
Output: Object
Semantics: Returns an object of the same type as the input (word or list) in which all alphabetic characters are transcribed in lower case and all other characters are returned unaltered.
Errors: None.
Note: Not in LOGO-S version 0.

7.3.4.7 Sort

Syntax: Sort FlatList
Inputs: Flat list
Output: List
Semantics: Returns a list of the elements of FlatList sorted alphabetically such that each elements is GreaterEq? than the next element.
Errors: List expected, word found.
List argument in wrong format.
Note: Not in LOGO-S version 0.

7.3.4.8 FullWriteForm

Syntax: FullWriteForm Object
Inputs: Any object
Output: Word
Semantics: Returns the idempotent form of the input. FullPrinting an object is the same as Printing the FullWriteForm of the object. See section 7.9, page 103 on idempotency.
Errors: None.

7.3.4.9 RequestForm

Syntax: RequestForm InputWord

Inputs: Word

Output: List

Semantics: Returns the internal representation form of the input as if the input had been typed to Request. Returns the input word as a list, converting spaces to element separators, converting open and close square brackets to open and close sublist delimiters, converting character codes surrounded by ^^s to the specified characters, and eliminating redundant separators (multiple separators between elements, separators at the beginning or end of the list, and separators immediately preceded or followed by sublist delimiters). Because RequestForm eliminates redundant separators and because FullWriteForm returns quoted external brackets in transcribing a list these primitives are not simple inverses of each other, but they do bear the following relationship:

Object = First RequestForm FullWriteForm Object

See Request, page 107 and also section 7.9 on idempotency, page 104.

Errors: Word expected, list found.

Ill-formed list object encountered.

Example:

```
FPrint LW :Space "A :Space :Space "B
 A  B
FPrint RequestForm LW :Space "A :Space :Space "B
[A B]
```

Note that the input to RequestForm was a five element word containing three spaces, and that the output was a 2-element list.

Note: Unmatched square brackets in InputWord will cause an error.

7.4 Arithmetic

7.4.1 Types of numbers

In a LOGO command line, a word which is not prefixed by " (or protected within lists) is tested to see if it can be converted to a number. If it can be, it is stored as a normalized number.
LOGO-S recognizes three types of numbers: integers, floating-point numbers and fixed-point numbers, but only integer arithmetic will be available in LOGO-S version 0.

Numbers may be typed in a variety of formats. Internally, they are stored in normalized form. (Number strings that are protected by quotes or brackets are immune to normalization and therefore can be preserved in arbitrary form.) Arithmetic operations return numbers only in normalized form, and furthermore, restrict the decimal portion of a fixed or floating point number to :Precision places. (However, at least 3 extra digits of hidden precision are maintained for computational purposes). Commands and operations which treat their inputs as strings instead of numbers (except **FullWriteForm** and its derivatives, e.g., **FullPrint**) will convert numbers to strings no longer than **:Precision** digits, regardless of the number of places stored. This is done by the **Print** command. **FullPrint** shows all digits, regardless of precision. In particular, it converts and shows digits of hidden precision. Additional zeros are not appended in low order positions to the right of the decimal point when fewer digits of precision are specified in the number than are permitted by **:Precision**.

Normalization

- omits the plus sign on positive numbers and exponents
- omits all signs on zero
- omits leading zeros to the left of integers, integer portions of fixed and floating point numbers, and exponents in floating point numbers
- fixes one digit (non-zero) to the left of the decimal point in floating-point numbers
- fixes a zero to the left of the decimal point in fixed-point numbers with absolute value between 0 and 1.

Definitions of the three types follow (normalization restrictions specific to the types are repeated in the definitions of each type along with specifications for input and internal representation):

Integers On type-in, these are strings of characters consisting of an optional sign (+ or -) followed by an arbitrarily long number of digits with no intervening spaces, punctuation or other symbols (such as $ or #) of any kind.

Normalized form omits leading zeros, the plus sign everywhere and all signs on zero.

Fixed point numbers

On type-in, these are strings of characters consisting of an integer portion (as defined above) followed by a decimal point followed by an arbitrarily long string of digits. Digits on either side of the decimal point (but not both) may be omitted.

If the number of digits to the right of the decimal point is greater than :Precision on type-in, that precision is maintained, though only :Precision plus the extra stored digits can be used in computations.

Normalization (explicit or through use as a number) follows rules for integers and provides a zero to the left of the decimal point if no digits were typed there by the user (or if an arithmetic operation generates a result between -1 and 1).

Floating point numbers

On type-in, these are strings of characters consisting of a fixed-point number (as defined above), an e (upper or lower case), and an integer (as defined above) to represent the exponent.

Normalization (explicit or through use as a number) removes plus signs affixed either as the first character of the string or as the first character following the e. Normalization also assures that exactly one digit, non-zero, precedes the decimal point. The e is rendered in lower case.

Arithmetic operations return results in which the total number of significant digits does not exceed :Precision plus the extra stored digits. Print shows :Precision significant digits and arbitrary length exponents.

7.4.2 Basic arithmetic operations

All basic arithmetic operations described in this subsection (except Sqrt) return arithmetic results in a format dependent on their inputs. If both inputs are in integer format, the result is returned in integer format. If either input is in floating point format, the result is returned in fixed point format. Otherwise the result is returned in fixed point format.

All infix binary operations must be surrounded by spaces. The unary minus operator affix form must be preceded by a space, but differs from the binary minus operation in that it is not followed by a space.

7.4.2.1 Sum +

Syntax: Sum Num1 Num2
Inputs: Two numbers.
Output: Number
Semantics: Returns the sum of exactly two numeric args, Num1 and Num2.
Errors: Number expected, non-numeric object found.
Infix form: +
Greedy form: LongSum
Greedy Abbrev: LSum

7.4.2.2 Difference Diff -

Syntax: Difference Subtrahend Minuend
Synonym: Diff
Inputs: Two numbers.
Output: Number
Semantics: Returns the difference of the Subtrahend and Minuend. If either argument is floating point, the answer is expressed as a floating point number.
Errors: Number expected, non-numeric object found.
Infix form: -

7.4.2.3 Product *

Syntax:	Product Multiplicand Multiplier
Inputs:	Two numbers.
Output:	Number
Semantics:	Returns the product of the two inputs.
Errors:	Number expected, non-numeric object found.
Infix form:	*
Greedy form:	LongProduct
Greedy Abbrev:	LProd

7.4.2.4 Quotient /

Syntax:	Quotient Dividend Divisor
Inputs:	Two numbers.
Output:	Number
Semantics:	Returns the quotient of Dividend and Divisor. Integer division is truncated.
Errors:	Number expected, non-numeric object found. Division by zero.
Infix form:	/

7.4.2.5 Remainder Rem \

Syntax:	Remainder Dividend Divisor
Inputs:	Two numbers.
Output:	Number
Semantics:	When inputs are both integers, returns the remainder of the integer division (remainder is always the same sign as the dividend). This function is defined for all real numbers as shown below.

LOGO definition:

```
to Remainder [D'nd D'sor]
   Return :D'nd - :D'sor * Int ( :D'nd / :D'sor )
```

Errors:	Number expected, non-numeric object found. Argument out of range.
Synonym:	Rem
Infix form:	\
Note:	Remainder's behavior with non-integer inputs is designed to facilitate the writing of extended precision division using the Int of the Quotient of the two inputs.

7.4.2.6 Modulo Mod

Syntax: Modulo Number Modulus

Inputs: Two numbers.

Output: Number

Semantics: When inputs are both positive integers, returns *Number* mod *Modulus*. This function is defined for all real numbers as shown below.

LOGO definition:

```
to Modulo [Num Mod]
   Return :num - :mod * Floor ( :num / :mod )
```

Note: Modulo's behavior with non-integer inputs is designed to maximize closure in modular arithmetic applications with the exception that non-integer remainders are not resolved to their integer equivalence class (to prevent accidental poisoning of data by rounding error). Thus, though the mod 5 quotient of 3 and 2 is meaningful and is defined as 4, Mod (3. / 2) 5 returns 1.5 rather than 4. This provides information which can be used to write procedures that handle non-integers as the user chooses. Note also that Mod (3 / 2) 5 returns 1 because the quotient is evaluated first and is an integer quotient.

Errors: Number expected, non-numeric object found.
Argument out of range.

Synonym: Mod

Note: Not in LOGO-S version 0.

7.4.2.7 Negative Neg

Syntax: Negative Number
Synonym: Neg
Inputs: Number
Output: Number
Semantics: Unary minus, the negation operator. Returns the arithmetic negation of its input.
Errors: Number expected, non-numeric object found.
Note: An attached -, affixed directly (without spaces) to any numerical expression, returns the arithmetic negative of the expression. Thus,

```
-:n    and    -(0 + :n)    and    Negative :n
```

all return the same value. Note that a minus sign followed by a space is interpreted as the binary minus operator.

7.4.2.8 Sqrt

Syntax: Sqrt Number
Inputs: Non-negative number
Output: Non-negative number
Semantics: Returns the square root of its input.
Errors: Number expected, non-numeric object found.
Argument out of range.
Note: Not in LOGO-S version 0.

7.4.3 Trigonometric and Exponential operations

Note: All trigonometric functions express angles and interpret angles to be expressed in degrees.

7.4.3.1 Sin

Syntax: Sin Angle
Inputs: Number
Output: Floating point number
Semantics: Returns the sine of Angle.
Errors: Number expected, non-numeric object found.
Note: Not in LOGO-S version 0.

7.4.3.2 Cos

Syntax: Cos Angle
Inputs: Number
Output: Floating point number.
Semantics: Returns the cosine of Angle.
Errors: Number expected, non-numeric object found.
Argument out of range.
Note: Not in LOGO-S version 0.

7.4.3.3 Tan

Syntax: Tan Angle
Inputs: Number
Output: Floating point number.
Semantics: Returns the tangent of Angle.
Errors: Number expected, non-numeric object found.
Argument out of range.
Note: Not in LOGO-S version 0.

7.4.3.4 Arctan

Syntax: Arctan Object
Inputs: Number or flat list of two numbers.
Output: Floating point number.
Semantics: Returns the arctangent of its numerical input. If input is a list of two numbers X and Y, Arctan returns the arctangent of Quotient Y X (or 90, if X is 0).
Errors: Number expected, non-numeric object found.
List argument in wrong format.
Note: Not in LOGO-S version 0.

7.4.3.5 ln

Syntax: ln x
Inputs: Positive number
Output: Floating point number
Semantics: Returns the natural logarithm of x.
Errors: Argument out of range.
Note: Not in LOGO-S version 0.

7.4.3.6 Exp

Syntax: Exp Power
Inputs: Number
Output: Positive number
Semantics: Returns the value e raised to the power Power.
Errors: Number expected, non-numeric object found.
Argument out of range.
Note: Not in LOGO-S version 0.

7.4.4 Random numbers

7.4.4.1 Random

Syntax: Random LowBound HighBound
Inputs: Two objects which must each be a number or a flat list containing a single number.
Output: Number
Semantics: Returns a uniformly distributed random value bounded (inclusive) by the inputs. If both inputs are integers, returned value is an integer. If either input is not an integer, continuous range is assumed. If input is a list, bound is interpreted not to be inclusive. Type of output is as in basic arithmetic operations.
Errors: Number expected, non-numeric object found.
List argument in wrong format.
Note: When the range specified by the inputs is very large, values returned may not be fully continuous over the range (and within the domain, whether integer or real), and will depend upon the precision of the machine.
Not in LOGO-S version 0.

7.4.4.2 RandomDigit

Inputs: None.
Output: Single digit
Semantics: Returns a random digit.

7.4.4.3 SetRandomSeed Command

Syntax: SetRandomSeed seed
Inputs: Non-negative integer
Semantics: Initializes random number generator with seed. If seed is 0, generator selects a seed.
Errors: Number expected, non-numeric object found.
Argument out of range.

7.4.4.4 :RandomSeed

Contains: A list of two integers. The first of the list is the value which will initialize the random number generator to its current state. The last of the list will initialize the random number generator to the state in which it started this session.

7.4.5 Formatting primitives

7.4.5.1 Normalize Norm

Syntax: Normalize Number
Synonym: Norm
Inputs: Number
Output: Number
Semantics: Returns a normalized form of the input number. High order zeros in integers are removed, zero is returned without a sign, floating point numbers are represented with one digit to the left of the decimal point.
Errors: Number expected, non-numeric object found.
Note: Not in LOGO-S version 0.

7.4.5.2 Integer Int

Syntax: Integer Number
Synonym: Int
Inputs: Number
Output: Integer
Semantics: Returns the normalized integer portion of the numerical input. If the input is positive, returns the greatest integer less than the input; if negative, returns the least integer greater than the input.
Errors: Number expected, non-numeric object found.

7.4.5.3 Floor

Syntax: Floor Number
Inputs: Number
Output: Integer
Semantics: Returns the normalized form of the greatest integer less than the input.
LOGO definition:

```
to Floor [n]
   if :n = Int :n, return Int :n
   if :n < 0, return Int (:n - 1)
   return Int :n
```

Errors: Number expected, non-numeric object found.
Note: Not in LOGO-S version 0.

7.4.5.4 Fix

Syntax: Fix Number
Inputs: Number
Output: Fixed point number
Semantics: Returns its input in normalized fixed-point form, restricted by :Precision.
Errors: Number expected, non-numeric object found.
Note: Not in LOGO-S version 0.

7.4.5.5 Float

Syntax: Float Number

Inputs: Number

Output: Floating point number.

Semantics: Returns its input in normalized floating-point form, restricted by :Precision.

Errors: Number expected, non-numeric object found.

Example:

```
>print float -53
-5.3e1
>print float 4e-2
4.0e-2
>print float 31.4
3.14e1
```

Note: Not in LOGO-S version 0.

7.4.5.6 SetPrecision

Command

Syntax: SetPrecision [Fixed Float]

Inputs: A list containing two numbers.

Output: None.

Semantics: Sets the precision to be used in fixed and floating point computations. For fixed point arithmetic, computations are rounded to Fixed decimal places. Floating point arithmetic rounds to Float significant figures. (See Precision, page 67)

Errors: Number expected, non-numeric object found.

Note: Not in LOGO-S version 0.

7.4.5.7 :Precision

Contains: A list of two integers which are the precision values used for fixed and floating point computations.

Note: Not in LOGO-S version 0.

7.4.5.8 Format

Syntax: Format InputNumber FormatSpecList

Inputs: A number and a flat list of positive integers.

Output: A word possibly containing leading <space>s.

Semantics: Returns a representation of InputNumber formatted for tabular presentation. (I.e., numbers Printed in a column and Formatted with the same FormatSpecList will "line up" properly.) Format first converts InputNumber to the desired type (integer, fixed, or floating) and then adds spaces and/or #-characters, if necessary, to format for columnar presentation.

FormatSpecList must contain from 1 to 3 positive integers: reserved length (L), local precision (P) and exponent digits (E), respectively.

- If only L is specified, Format converts InputNumber to an integer (by rounding rather than truncation). L specifies the minimum length (in characters) of the word returned by Format.

- If L and P (but not E) are specified, Format converts InputNumber to a fixed point number rounded to P decimal places. P indicates the maximum number of decimal places to be printed. If P exceeds :Precision, spaces are appended.

- If E is specified, Format converts InputNumber to a floating point number rounded to P decimal places, forces two characters (either "#e" or "e-", depending on the sign of the exponent) after the final decimal digit of the result and designates E as the maximum number of digits that the exponent may contain. If P exceeds :Precision, #'s are inserted before the "e".

If Count of the exponent is less than E, #'s are inserted before the "e". If Count of the formatted number thus far is less than L, spaces are prepended to the word to make up the difference. If either E or L are exceeded, no formatting changes (additions or truncations) are performed. If specifications are not exceeded, Format returns a word whose Count is L. In summary, format performs only three kinds of adjustments:

- Normalizing and rounding to P places

- Adding spaces to the left of any number or to the right of fixed point numbers

- Inserting #'s to the left of the "e" in floating point numbers

Note: If P is specified, L should not be less than P + 2 to account for a decimal point and at least one digit to the left of it. If E is specified, L should not be less than P + E + 4, to account for the "#e" or "e-".

If L is exceeded (due to too many digits in the integer portion of the number or in the exponent), no error is caused; the output expands to accommodate the need.

Note: Not in LOGO-S version 0.

Errors: Number expected, non-numeric object found.
List expected, word found.
List argument in wrong format.

Example:

InputNumber in normal print format	FormatSpecList and corresponding printed results		
	[8]	[8 1]	[8 1 2]
-0.000000000979	0	0.0	-9.8e-10
62	62	62.0	6.2##e1
1.73	2	1.7	1.7##e0
1.2e-1	0	0.1	1.2#e-1
-123456.5	-123457	-123457.0	-1.2##e5
123456789	123456789	123456789	1.2##e8
-12345678	-12345678	-12345678	-1.2#e-7
1.3e350	13000000000*	13000000000*	1.3#e350

*Starred numbers above have, in reality, 351 digits. Underlined numbers have exceeded specifications.

7.5 Procedures, Names and Arrays

7.5.1 Defining procedures

7.5.1.1 To

Command

Syntax: To ProcedureName [args list] [local vars list]

Inputs: One, two or three literal inputs, the first of which must be a word, and the others of which, if present, must either be empty or lists that contain words only. ProcedureName must be a word and may not be a number nor may it begin with : or the " mark. Contents of the second and third inputs (if any) are the names of dummy arguments to the procedure and local variables, respectively. If there are no local variables, the third input may be omitted. If there are neither local variables nor arguments, both the second and third inputs may be omitted. If there are local variables but no arguments, all three inputs are required and the empty list is used to designate the arguments.

Semantics: To is used to define a "normal" procedure, i.e., a procedure which expects a fixed number (zero or more) of evaluated arguments, corresponding to the specified variables in the args list. Local variables (if any) are undefined when the procedure is invoked and may be defined during execution of the procedure. When the procedure terminates, all variables are restored to the values they had prior to procedure invocation.

To checks its input to determine if procedure ProcedureName already exists. If it does, To notifies the user and quits. If no such procedure exists, To enters edit-mode to allow user to specify the text of the procedure (see Edit, page 73).

Errors: Word expected, list found.
Non-numeric word expected, number found.
Procedure already exists
List argument in wrong format.
(To checks its inputs.)

7.5.1.2 ToGreedy Command

Syntax:	ToGreedy ProcedureName [ArgName] [local vars list]
Inputs:	Two or three inputs, the first of which must be a word, the second of which must be a list containing exactly one word, and the last of which, if present, must be empty or contain words only. ProcedureName and ArgName must be words. For consistency with To, ArgName must be enclosed in brackets. Contents of the (optional) third input to ToGreedy are the names of local variables.
Semantics:	ToGreedy is used to define a "greedy" procedure, i.e., a procedure which, on invocation, permits a variable number (zero or more) of evaluated arguments. On invocation, all the evaluated inputs to the right of the procedure call (terminated by the end of the logical line, by a semicolon, or by a right parenthesis balancing one that immediately preceded the procedure name) will be collected as a list. The name of this list must be specified at the time of definition as the single word ArgName in the required second input to ToGreedy. In other respects, ToGreedy functions like To. See To, page 70.
Errors:	Word expected, list found. Non-numeric word expected, number found. Procedure already exists List argument in wrong format. (ToGreedy checks its inputs.)
Note:	Sum and List are examples of primitives which are greedy.

7.5.1.3 ToQuoting

Command

Syntax: ToQuoting ProcedureName [arg list] [local vars list]

Inputs: Two or three inputs, the first of which must be a word, the second of which must be a list of words, and the last of which, if present, must be empty or contain words only. Their interpretation and other restrictions are identical with **To**, above.

Semantics: **ToQuoting** is used to define a "quoting" procedure, i.e., a procedure which, on invocation, does not evaluate its arguments. In other respects, **ToQuoting** functions like **To**. See **To**, page 70.

Errors: Word expected, list found.
Non-numeric word expected, number found.
Procedure already exists
List argument in wrong format.
(**ToQuoting** checks its inputs.)

Note: **Restore** and **Edit** are examples of primitives which are quoting.

7.5.1.4 Define

Command

Syntax: Define ProcedureText

Inputs: List of lists

Semantics: Interprets its input as the text of a procedure to be defined. The first sublist of ProcedureText should contain a title line beginning with the word **To**, **ToGreedy**, or **ToQuoting** and continuing with the appropriate syntax. Remaining sublists each contain two words, the first of which is a line number (integer) and the second of which is the text of the corresponding logical line. These lists must be in ascending order by line number. (The list returned by **Text** is in the proper format to be input to **Define**.) If there exists a procedure with the same title, the procedure's definition is replaced with the new text. Otherwise, a new procedure is created.

Errors: List argument in wrong format.
Object protection violated.

7.5.1.5 NoText

Command

Syntax:	NoText ProcedureName
Inputs:	A word
Semantics:	The specified procedure is marked to have its source text erased from the workspace. Erasing of source text takes place only after the procedure has been parsed successfully, i.e., after the first invocation that does not cause a parse error. Once text is removed, a procedure cannot be Saved or Edited and only their title lines can be Shown (i.e, Text returns only a title line).
Errors:	Word expected, list found. Non-numeric word expected, number found. Non-existent object. Object protection violated.
Note:	There is no recovering the text of a NoTexted procedure unless that text was previously saved. Autosaving of text and autorecovery of workspace is anticipated in later versions of LOGO-S.

7.5.1.6 Text?

Syntax:	Text? ProcedureName
Inputs:	A word
Output:	A Boolean value.
Semantics:	Returns True if the text of the specified procedure has been retained in the workspace, False otherwise.
Errors:	Word expected, list found. Non-numeric word expected, number found. Non-existent object. Object protection violated.

7.5.2 Editing procedures

7.5.2.1 Edit

Ed **Command**

Syntax: Edit ProcedureName

Inputs: Unquoted word

Semantics: Enters edit-mode, enabling the editing commands to modify the text of the specified procedure, ProcedureName. When edit mode is terminated (using **End**), the edited copy of the procedure replaces the previous definition. If changes have been made, the message "Procedure ProcedureName replaced" is printed at the terminal. (But see **End** and **Restore**.)

Errors: Non-existent object.
Object protection violated.
Word expected, list found.
Non-numeric word expected, number found.

Note: All editing takes place on a copy of the procedure. This copy is in the form returned by **Text** and is kept in **"#EditBuffer**. Edit quotes its argument.

Synonym: **Ed**

7.5.2.2 TitleLine

Syntax: TitleLine ProcedureName

Inputs: Word

Output: List

Semantics: Returns a list whose elements correspond to the 2-4 elements of the title line of the specified procedure: respectively, one of **To**, **ToGreedy**, or **ToQuoting**; the procedure name; the argument list (present only if the procedure takes arguments or uses local variables); and the local variables list (present only if the procedure uses local variables).

Errors: Word expected, list found.
Non-existent object.
Object protection violated.
Non-numeric word expected, number found.

7.5.2.3 TextLine

Syntax:	TextLine ProcedureName LineNumber
Inputs:	A word and a positive integer
Output:	List
Semantics:	Returns a list of two elements. The first element is LineNumber and the second element is a word containing the text of that line of the specified procedure. Returns empty if ProcedureName is non-existent, if LineNumber does not exist within ProcedureName or if the text of that line is not available.
Errors:	Word expected, list found. Non-numeric word expected, number found. Number expected, non-numeric object found.

7.5.2.4 Lines

Syntax:	Lines ProcedureName
Inputs:	A word
Output:	List
Semantics:	Returns a list of the line numbers in the specified procedure.
Errors:	Word expected, list found. Non-numeric word expected, number found.

7.5.2.5 Text

Syntax:	Text ProcedureName
Inputs:	Word
Output:	List
Semantics:	Returns list of lists containing the text of specified procedure. The first sublist of ProcedureText contains a title line beginning with the word To, ToGreedy, or ToQuoting and continuing with the remaining text of the title line (as returned by TitleLine). The remaining sublists each contain two words, the first of which is a line number (integer) and the second of which is the text of the corresponding line (as returned by TextLine). Returns the empty list if ProcedureName is not defined or has the property "Invisible. Returns a title line only if ProcedureName is a primitive or has the properties "Primitive or "NoText.
Errors:	None.

7.5.2.6 Replace Command

Syntax: Replace Line

Inputs: A list (such as returned by **TextLine**) containing two elements, an integer line number, and the word containing the text of that line respectively.

Semantics: Meaningful only in edit mode. Creates the specified line in the procedure being edited.

Note: **Replace S** Num **L TextLine** ProcName Num2 can be used to copy text from a procedure into the edit buffer.

7.5.2.7 EditLine Edl Command

Syntax: EditLine LineNumber

Inputs: Integer

Semantics: Enables the user to edit the specified line (of the edit buffer) using the special editing characters (see **EditString**, 107). When **:EOL** is typed, the edited copy of the line is replaced in the edit buffer.

Errors: Number expected, non-numeric object found.

Synonym: **Edl**

7.5.2.8 EditTitle Edt Command

Inputs: None.

Semantics: Enables the user to edit the title line (of the procedure copy in the edit buffer) using the special editing characters (see **EditString**, 107). When **:EOL** is typed, the edited title is replaced in the edit buffer.

Synonym: **Edt**

7.5.2.9 Restore Old Command

Syntax:	Restore Scope
Inputs:	Scope may be an integer or list of integers, or may be the (unquoted) word **TITLE** or the name of the procedure that was used as an argument to **Edit** (i.e., ProcedureName).
Semantics:	Meaningful only inside an edit mode that was entered via **Edit**. If Scope is the word **TITLE**, the title line is restored to the text it had when editing was begun. If Scope is an integer or list of integers, text with line numbers in that range is restored to its old state. The user is notified if lines were previously undefined, but those lines are not erased. If Scope is the old procedure´s name (unquoted), all text of the procedure being edited is restored to its original state. If **End** is executed immediately after **Restore** ProcedureName, edit mode is exited without modifying the procedure being edited, and the message "Procedure ProcedureName Unchanged" is printed.
Errors:	None. Restore checks its inputs.
Note:	Restore quotes its input.
Synonym:	Old

7.5.2.10 End Command

Inputs:	None.
Semantics:	Meaningful only in edit mode. Exits edit mode and makes the new or newly modified procedure part of procedure workspace. Causes message to be printed indicating whether procedure has been newly created ("procedure defined") or modified from a previous definition ("procedure changed").

7.5.3 Names

7.5.3.1 Make <- Command

Syntax: Make Name Thing
Inputs: Word and object
Semantics: Binds the value Thing to the name Name.
Errors: Word expected, list found.
Non-numeric word expected, number found.
Object protection violated.
Infix form: <-

7.5.3.2 Thing

Syntax: Thing Name
Inputs: Word
Output: Object
Semantics: Returns the value currently bound to the name. Error if there is no such value.
Errors: Word expected, list found.
Non-numeric word expected, number found.
Undefined name.
Object protection violated.
Note: The : affixed to a name (e.g., :Name) has the same effect as Thing "Name. See :, page 136.

7.5.4 Arrays

Array primitives will be further developed and extended in later versions of LOGO-S.

7.5.4.1 DefineArray — Command

Syntax:	DefineArray ArrayName DimensionList
Inputs:	A word, a list and an arbitrary object
Semantics:	Creates array with specified name and dimensions. The name may be any non-numeric word. DefineArray checks its input to determine if array ArrayName already exists. If it does, DefineArray notifies the user and quits. The DimensionList must be in the following format: [DimSpec1 DimSpec2 ... DimSpecN] Each dimension specification (DimSpec1, etc.) in DimensionList may either be an integer or a two-element list of integers. If a list, the first element specifies the lowest value of that dimension and the second value specifies the highest. If an integer, it specifies the highest value of that dimension and a default low value of 0 is assumed.
Errors:	Word expected, list found. Non-numeric word expected, number found. List expected, word found. List argument in wrong format. Argument out of range.
Note:	Not in LOGO-S version 0. Initialization of array cells will be specified subsequently.

7.5.4.2 Store

Command

Syntax:	Store Object ArrayName IndexList
Inputs:	A word or list, a word, a list
Output:	None.
Semantics:	Stores *Object* in array *ArrayName* at the location specified by *IndexList*.
Errors:	Word expected, list found. Non-numeric word expected, number found. List expected, word found. Argument out of range. List argument in wrong format. Object protection violated.
Note:	Not in LOGO-S version 0.

7.5.4.3 Fetch

Syntax:	Fetch ArrayName IndexList
Inputs:	Word, List
Output:	Object
Semantics:	Returns object contained in array *ArrayName* at the location specified by *IndexList*.
Errors:	Word expected, list found. Non-numeric word expected, number found. List expected, word found. Argument out of range. List argument in wrong format. Object protection violated.
Note:	Not in LOGO-S version 0.

7.6 The workspace

When LOGO-S is started up, the user is provided with the use of all of the primitives and names described in this glossary and a "workspace" in which to create new structures -- procedures, names (variables), and arrays. These structures may be manipulated in various ways, e.g., they may be Shown at the user's terminal, Erased from the workspace, Traced, etc. To aid in the manipulation of these structures, LOGO provides "workspace descriptor lists," which describe all or parts of the current workspace. These lists contain one or more pairs of elements, the first of which is a descriptor (one of **PROCEDURES**, **NAMES**, **ARRAYS**, or **FILES**) and the second of which is a list containing the names of workspace constituents of that type.

Example:

```
            [PROCEDURES [Circ Arc] NAMES [Rad] PROCEDURES [Draw]]
```

Workspace descriptor lists are frequently the desired inputs to primitives such as **Show**, **Erase**, **Trace**, and **Untrace**.

7.6.1 Contexts

In later releases of LOGO-S (but not in version 0), the concept of workspace **Contexts** will be developed to allow multiple workspaces. (These will be useful for parallel processing and other context-sensitive applications.) In these versions, the initial workspace will have the context name "Common." Contexts created within other contexts will inherit the parent context properties, but may contain, locally, their own constituents as well. For example, procedures defined within "Common" will be available for use in all of its member contexts. Procedures defined (or redefined) in member contexts will only be available in those contexts or in their descendants. The same will hold true of workspace constituents other than procedures, and for properties of constituents, trap-handling states, etc.

Primitives for defining and undefining contexts, moving from one to another, and transfering constituents or values from one to another will be developed in later releases of LOGO-S.

7.6.2 Workspace Constituents

7.6.2.1 Procedures Procedure

Syntax: Procedures ProcName
Inputs: Word or flat list
Output: Workspace descriptor list
Semantics: Returns a workspace descriptor list whose first element is the word **PROCEDURES**. If **ProcName** is a list, the second element of the output list will be **ProcName**; if **ProcName** is a word, the second element will be a list containing **ProcName**.
Synonym: **Procedure**
Errors: List argument in wrong format.
Example:

```
>FullPrint Procedure "Foo
[PROCEDURES [Foc]]
>FullPrint Procedures [Foo BaR]
[PROCEDURES [Foo BaR]]
```

7.6.2.2 Names **Name**

Syntax: Name ThingName
Inputs: Word or flat list
Output: Workspace descriptor list
Semantics: Returns a workspace descriptor list whose first element is the word **NAMES**. If **ThingName** is a list, the second element of the output list will be **ThingName**; if **ThingName** is a word, the second element will be a list containing **ThingName**.
Synonym: **Name**
Errors: List argument in wrong format.

7.6.2.3 Arrays	**Array**
Syntax:	Array ArrayName
Inputs:	Word or flat list
Output:	Workspace descriptor list
Semantics:	Returns a workspace descriptor list whose first element is the word **ARRAYS**. If **ArrayName** is a list, the second element of the output list will be **ArrayName**; if **ArrayName** is a word, the second element will be a list containing **ArrayName**.
Synonym:	**Array**
Errors:	List argument in wrong format.

7.6.2.4 Files	**File**
Syntax:	File FileName
Inputs:	Word or flat list
Output:	Workspace descriptor list
Semantics:	Returns a workspace descriptor list whose first element is the word **FILES**. If **FileName** is a list, the second element of the output list will be **FileName**; if **FileName** is a word, the second element will be a list containing **FileName**.
Synonym:	**File**
Errors:	List argument in wrong format.

7.6.2.5 All	
Syntax:	All KeyWord
Inputs:	**Keywords Procedures, Names, Arrays,** and **Files** only.
Output:	Workspace descriptor list
Semantics:	Returns a workspace descriptor list whose first element is the keyword and whose second element is a list containing names of all workspace structures matching that keyword, provided that these structures are not marked with the property "Invisible. In the case of **Files**, all open files are returned. (See "Invisible, page 118.) Useful after primitives **Erase**, **Show**, **Trace** and **Untrace**, e.g., **Show All Names**
Errors:	Unknown keyword in this context

7.6.2.6 Invisible

Syntax: Invisible KeyWord

Inputs: Keywords Procedures, Names, Arrays, Files, and Workspace only.

Output: Workspace descriptor list

Semantics: Returns a workspace descriptor list enumerating contents of the workspace corresponding to the keyword and marked with the property "Invisible. (See "Invisible, page 118.)

Errors: Unknown keyword in this context

7.6.2.7 Primitive

Syntax: Primitive KeyWord

Inputs: Keywords Procedures, Names, only.

Output: Workspace descriptor list

Semantics: Returns a workspace descriptor list enumerating contents of the workspace corresponding to the keyword and marked with the property "Primitive. Show Primitive Procedures will display a list of the title lines of all primitives not marked "Invisible. Similarly Show Primitive Names will display all system protected names and their values, unless the names have the additional property "Invisible. Primitive procedures and names can be listed and Shown, but cannot be Erased, Saved or otherwise manipulated. No properties of primitive structures can be modified except by the implementer. (See "Primitive, page 118.)

Errors: Unknown keyword in this context

7.6.2.8 WorkSpace

Inputs: None.

Output: Workspace descriptor list

Semantics: Returns a workspace descriptor list enumerating all visible contents of the workspace, procedures first, names second and arrays third. (See definition of descriptor list, page 81 and "Invisible, page 118.)

7.6.3 Workspace Manipulators

7.6.3.1 Erase Command

Syntax: Erase Specifier

Inputs: Specifier must be

Case 1. a quoted word which names a procedure.

Case 2. a flat list -- interpreted as a list of procedure names

Case 3. a workspace descriptor list (see 81)

Semantics: Erases the specified structure or structures from the workspace.

Errors: Non-existent object.
Object protection violated.
List argument in wrong format.

Synonym: Er

Note: Erase may be used with procedures, names or arrays. Delete is used for eliminating SaveFiles.

7.6.3.2 Show Sh Command

Syntax: Show Specifier

Inputs: In edit-mode, Specifier may be

Case 1. <omitted> -- indicates structure to be shown is the edit buffer copy of the procedure being defined or edited

Case 2. <integer> -- refers to line <integer> of the edit buffer copy of the procedure being defined or edited

Case 3. list containing pair of integers -- refers to lines (numbered within range of the input pair, inclusive) of the edit buffer copy of the procedure being defined or edited

In or out of edit-mode, Specifier may be

Case 1. a quoted word which names a procedure.

Case 2. a flat list of non-numeric words -- interpreted as a list of procedure names

Case 3. a workspace descriptor list (see page 81)

Case 4. a list containing a list as its first element

Semantics: Types at the terminal an appropriately formatted representation of the structures specified in the input.

Errors: Non-existent object.
Object protection violated.
List argument in wrong format.
Non-numeric word expected, number found.

Note: The formatting of a list containing a list as its first element is, simply, one element per line. This permits Show to be used in connection with Title and Titles.

Note: Confirmation by typing Yes is required if more than one name appears in a FILES workspace descriptor sublist.

Synonym: Sh

7.6.3.3 Title

Syntax: Title Specifier

Inputs:

Case 1. Specifier may be omitted in edit mode, and default is title of procedure in edit buffer.

Case 2. Quoted word, indicating a procedure name

Case 3. A flat list, interpreted as a list of procedure names

Output: A list of lists.

Semantics: Returns a list of sublists containing the title lines of all visible specified procedures in the user workspace, one title per sublist.

Errors: List argument in wrong format.
Non-numeric word expected, number found.

7.6.3.4 Titles

Ts

Inputs: None.

Output: A list of lists.

Semantics: Returns a list of sublists containing the title lines of all visible procedures in the user workspace, one title per sublist.

Synonym: Ts

7.7 Control structure

7.7.1 Concepts of control structure in LOGO

LOGO-S version 0 provides capabilities for recursive and iterative control structures. Capabilities for parallel processing, coroutines, case structures, class-instance, and map functions, will be specified in later versions of LOGO-S.

7.7.2 Conditional Statements

7.7.2.1 If — Control Word

Syntax: If Boolean CondDelimiter StatementSeq

Semantics: Boolean must either be or evaluate to True or False. CondDelimiter is a control delimiter -- one of do, then, or , -- separating the predicate expression from the statement sequence that is to be executed if the predicate evaluates to True. StatementSeq may optionally be followed by the control delimiter else, followed by a second statement sequence that is intended to be executed if the predicate evaluates to False. Statement sequences are either single LOGO statements or multiple statements separated by semi-colons. If, While, and Until statements may be nested to arbitrary depth within statement sequences.

Errors: Misplaced syntactic marker: Expecting Then, found....
Boolean expected, non-Boolean found.

Note: When an If statement follows a Then (or its synonyms, , and Do), the If statement´s full scope must be delimited by curly braces.

7.7.2.2 Then Do , — Conditional Delimiter

Syntax: ControlWord Boolean Then StatementSeq

Semantics: Syntactical marker between Boolean condition and statement sequence in conditional statements. Synonym for , and Do.

Errors: Misplaced syntactic marker: Expecting..., found Then.

Synonym: Do

Synonym: ,

7.7.2.3 Do

Then , Conditional Delimiter

Syntax: ControlWord Boolean Do StatementSeq

Semantics: Syntactical marker between Boolean condition and statement sequence in conditional statements. Synonym for , and Then.

Errors: Misplaced syntactic marker: Expecting..., found Do.

Synonym: Then

Synonym: ,

7.7.2.4 ,

Then Do Conditional Delimiter

Syntax: ControlWord Boolean , StatementSeq

Semantics: Syntactical marker between Boolean condition and statement sequence in conditional statements. Synonym for Do and Then.

Errors: Misplaced syntactic marker: Expecting..., found ,.

Synonym: Then

Synonym: Do

7.7.2.5 Else

Conditional Delimiter

Syntax: If Boolean CondDelimiter StatementSeq1 Else StatementSeq2

Semantics: Used in If statements to precede a statement sequence StatementSeq2 that is to be executed if the condition evaluates to False.

7.7.2.6 While

Control Word

Syntax: While Boolean CondDelimiter StatementSeq

Semantics: Repeats execution of StatementSeq while Boolean evaluates to True.

Errors: Misplaced syntactic marker: Expecting Then, found....
Boolean expected, non-Boolean found.

7.7.2.7 Until

Control Word

Syntax: Until Boolean CondDelimiter StatementSeq

Semantics: Repeats execution of StatementSeq until Boolean evaluates to True.

Errors: Misplaced syntactic marker: Expecting Then, found....
Boolean expected, non-Boolean found.

7.7.2.8 Repeat

Control Word

Syntax: Repeat Number CondDelimiter StatementSeq

Semantics: Repeat execution of StatementSeq specified Number of times.

Errors: Misplaced syntactic marker: Expecting Then, found....
Number expected, non-numeric object found.

7.7.2.9 Go

Control Word

Syntax: Go LineNumber

Inputs: Positive integer

Semantics: Transfers control to line LineNumber of currently active procedure.

Errors: Number expected, non-numeric object found.
Non-existent object. (No such line)
Executed while not in procedure.

7.7.3 Returning from procedures

7.7.3.1 Return R

Command

Syntax: Return Object

Inputs: Any object

Semantics: Causes procedure which executes it to terminate and return the value Object. Return must be executed within a procedure and does not, itself, return a value.

Errors: Executed while not in procedure.

Synonym: R

7.7.3.2 Done

Command

Inputs: None.

Semantics: Causes procedure which executes it to terminate without returning a value.

Errors: Executed while not in procedure.

7.7.3.3 Unwind

Command

Syntax: Unwind Level

Inputs: Non-negative integer

Semantics: Until Level is equal to Level, causes procedure which executes it (and its calling procedure(s)) to terminate without returning a value.

Errors: Number expected, non-numeric object found.
Argument out of range.

Note: Attempts to unwind to levels higher than Level have no effect. Attempts to unwind to levels below the top level monitor will unwind only to that monitor. Attempts to unwind to levels not accessible to the user (e.g., levels generated by a Run) will unwind to the nearest lower accessible level.

7.7.3.4 UnwindWith

Command

Syntax: UnwindWith Level Value

Inputs: An integer and any object

Semantics: Operates as Unwind, unwinding the stack back to the procedure call at level Level, except that it causes Value to be returned to the calling procedure at that level.

Errors: Number expected, non-numeric object found.
Argument out of range.

Note: Attempts to unwind to levels higher than Level have no effect. Attempts to unwind to levels below the top level monitor will unwind only to that monitor. Attempts to unwind to levels not accessible to the user (e.g., levels generated by a Run) will unwind to the nearest lower accessible level.

7.7.3.5 Level

Inputs: None.

Output: Non-negative integer

Semantics: Returns the number of active stack frames, equal to the number of currently active command lines. (Generally, this is the same as the procedural nesting of the current procedure, but it is increased by Run, Value, calls to #Monitor and Suspend as well as by procedure calls.) If the monitor is at level n, the level inside a procedure called from that monitor is n+1, the level inside a procedure called by that procedure is level n+2, etc.

7.7.3.6 :TopLevel

Contains: Level of the entering (top level) monitor.

7.7.4 Evaluators

7.7.4.1 Run

Command

Syntax: Run Statement

Inputs: Any object

Semantics: Evaluates *Statement* as if it had been typed at the terminal. *Statement* must consist of a command or sequence of *commands* separated by semi-colons. See TrapReturn, page 99.

Note: Run is the LOGO procedure which invokes the parser and evaluator on all statement lines. Errors occurring within statements inside Run are attributable to the statements themselves and not to Run. Their location is reported as the procedure and line (or top level) which called Run. An error type specific to the Run environment is "Operation expected, command found."

7.7.4.2 Value

Syntax:	Value Expression
Inputs:	Any object
Output:	Object
Semantics:	Evaluates Expression as if it had been typed following a command (such as Print) at the terminal. Expression must consist of a single operation or a single expression (statement returning a value) composed of operations. See TrapReturn, page 99.
Note:	Value is the LOGO procedure which parses and evaluates LOGO expressions. Errors occurring within expressions inside Value are attributable to the expressions themselves and not to Value. Their location is reported as the procedure and line (or top level) which called Value. An error type specific to the Value environment is "Command expected, operation found."

7.8 Trapping, Error Handling and Debugging

The sophisticated interrupt handling capability of LOGO-S provides nine channels of interrupts covering error conditions, a clock trap, user initiated interrupts, and special conditional interrupts. A primary value of this design is in the development of helpful error diagnostics and intelligent debugging aids.

Traps for any of the nine conditions (explained below) may be set. When a condition occurs for which a trap has been set, a user-designed procedure is executed. (In some cases, including all error conditions and the panic stop, default handlers provided with LOGO-S are executed if there are no user procedures superseding them.) Information about the circumstances under which the interrupt occurred is passed to the trap handling procedure which may use it to decide, for example, how to recover after an error.

Trap types include

Allocator	Storage allocation is used up. (Error condition.)
Evaluator	Evaluation of a primitive has failed. (Error condition.)
File	I/O handling has failed. (Error condition.)
Parse	An expression cannot be parsed. (Error condition.)
Keyboard	Some particular (implementation dependent) key(s) on the keyboard have been depressed. (User-initiated condition.)
Panic	The panic stop control character sequence has been pressed (a user-initiated condition) or an interrupt has been generated on a sprung or unset channel (an error condition).
Timer	The timer has interrupted. (Special condition.)
Trace	A traced object has been accessed. (Special condition.)
Warning	(Primarily for SpringTrap.) A specified (generally, user-specified) condition has occurred. (Special condition.)

A trap handler procedure must be a normal procedure (i.e., defined by To) requiring exactly seven inputs corresponding to the seven items of information provided about the circumstances of the interrupt. When a condition occurs for which this procedure is the designated handler, the procedure will be invoked and its seven arguments set, respectively, to the TrapType, TrapObject, TrapContext, TrapLevel, TrapProcedure, TrapLine, and TrapCode. Although these values are described under the names just given, it should be noted that they are available only under the (user-chosen) names to which they correspond in the title line of the trap handler. Thus, if the title line is

```
        To .#Eval#. [Typ Obj Env Lvl Pro Lin Cod] [Em1 Em2]
```

then TrapContext would be available only as :Env and only while .#Eval#. is being executed.

The values passed to the handler are:

TrapType	A word indicating the type of trap. One of: ALLOCATOR, EVALUATOR, FILE, PARSE, KEYBOARD, PANIC, TIMER, TRACE, or WARNING.
TrapContext	Normally, a list containing a primitive followed by its (literal) arguments. A handler which runs **TrapReturn** with **TrapType** and **TrapContext** as its inputs will re-do the primitive that caused the error. Some modification to the contents of **TrapContext** may be made before **TrapReturn** is used.
TrapObject	If trap type is Case 1. **Evaluator, File, Trace** or **Warning:** A positive integer indicating position (in **TrapContext** list) of argument causing the trap Case 2. **Parse:** tokenized line up to token causing error Case 3. **Keyboard, Panic:** character causing interrupt Case 4. **Timer:** number of times trap has been sprung since start of session Case 5. **Trace:** object name Case 6. **Allocator:** empty If trap is sprung by **SpringTrap**, **TrapObject** contains whatever **SpringTrap** placed there.
TrapLevel	Level at which trap occurred. (See **Level**.)
TrapProcedure	Name of the procedure in which the trap occurred, or **Empty** if the trap occurred at top level.
TrapLine	Line number (of procedure) in which trap occurred, 0 while executing the title line of a procedure, or **Empty** if the trap occurred at top level.
TrapCode	An integer. Trap codes greater than zero designate the trap type. If **TrapCode** is zero, **TrapContext** contains the actual value that must be returned in order to proceed from the trap.

Setting a trap for a particular interrupt involves first defining the procedure that will handle the trap and then associating the trap type with that (already existing) procedure by means of the SetTrap command. SetTrap checks the procedure title (existence, normalcy, seven arguments) and, if acceptable, pushes the current trap handler for the designated type onto a trap stack and establishes the new handler for that type. One may return to the previous handler(s) by using PopTrap.

7.8.1 Commands

7.8.1.1 SetTrap — Command

Syntax:	SetTrap TrapType TrapHandler
Inputs:	Two words, the first of which names a trap type and the second of which names a procedure to handle that trap.
Semantics:	Establishes TrapHandler as the procedure to execute when the TrapType trap is sprung. Pushes the previous trap-handler binding onto a stack. When the specified trap is sprung (whether by a trap condition or explicitly by SpringTrap), the procedure TrapHandler is executed.
Errors:	Word expected, list found. Unknown keyword in this context Non-existent object. List argument in wrong format. Object protection violated.

7.8.1.2 PopTrap — Command

Syntax:	PopTrap TrapType
Inputs:	Word
Semantics:	Pops to previous trap handling procedure for specified trap type. Default trap handlers for LOGO-S cannot be popped.
Errors:	Word expected, list found. Unknown keyword in this context Object protection violated.

7.8.1.3 SpringTrap Command

Syntax:	SpringTrap TrapType TrapObject TrapContext TrapCode
Inputs:	Word, objects depending on trap type
Semantics:	Sets TrapObject, TrapContext and TrapCode as specified and springs designated trap, executing appropriate trap handling procedure.
Errors:	Word expected, list found. Unknown keyword in this context
Note:	If trap is already in sprung state, error information is printed at the terminal and a panic stop is generated.

7.8.1.4 ResetTrap Command

Syntax:	ResetTrap TrapType
Inputs:	Word
Semantics:	Resets designated trap from Sprung state to Set state.
Errors:	Word expected, list found. Unknown keyword in this context Non-existent object. (No trap handler.)

7.8.1.5 SetTimer Command

Syntax:	SetTimer Milliseconds
Inputs:	Positive integer
Semantics:	Sets timer with a time period (in milliseconds) at the end of which the TimerTrap will be sprung.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.8.1.6 ClearTimer Command

Inputs:	None.
Semantics:	Resets timer to 0 without springing TimerTrap.

7.8.1.7 TrapReturn

Command

Syntax: TrapReturn TrapType Expression

Inputs: A word (designating a trap type) and a LOGO expression (word or list)

Semantics: Executes a command or operation returns control to level at which trap was sprung. (If expression was an operation, returns the value of the evaluated expression.) Executes expression as substitute for failed primitive in trap context. See also Value, page 93 and Run, page 92.

Errors: Word expected, list found.
Trap not in progress.

Note: TrapReturn :TrapType :TrapContext is the normal way of attempting to retry a primitive which causes an error. Expression is evaluated in the environment in which it occurred (as if the trap had not occurred) except that TrapReturn will not spring Trace traps.

7.8.2 Operations

7.8.2.1 TrapSet?

Syntax: TrapSet? TrapType

Inputs: A word

Output: A Boolean value.

Semantics: Returns True if specified trap is in Set state, False otherwise (Sprung state or Unset state). If a trap condition occurs while the corresponding trap is in Set state, the current trap handler is invoked. If it occurs while the trap is in Sprung state, processing is halted and the stack is unwound to current virtual top level (nearest monitor). If it occurs while the trap is in Unset state, processing is halted and the stack is unwound to top level.

Errors: Word expected, list found.
Unknown keyword in this context

7.8.2.2 TrapHandler

Syntax:	TrapHandler TrapType
Inputs:	A word
Output:	A list
Semantics:	Returns the list of procedure names on the trap handler stack for specified trap type. The first procedure is the current handler. The last procedure on the list is the name of the default handler (monitor's original handler).
Errors:	Word expected, list found. Unknown keyword in this context

7.8.2.3 ErrorMessage

Syntax:	ErrorMessage Type Obj Context Level Proc Line Code
Inputs:	The objects returned by TrapType, TrapObject, TrapContext, TrapLevel, TrapProcedure, TrapLine, and TrapCode, respectively.
Output:	Word
Semantics:	Returns appropriate message; returns :Empty if no message is found.
Errors:	None. ErrorMessage checks its inputs

7.8.2.4 Timer

Inputs:	None.
Output:	Number
Semantics:	Returns number of milliseconds left on timer before TimerTrap is sprung.

7.8.3 Debugging

7.8.3.1 BackTrace

Inputs:	None.
Output:	List
Semantics:	Returns list of titles of all active procedure calls.

7.8.3.2 Debug

Command

Syntax:	Debug or Debug On or Debug Off
Inputs:	Optional Boolean value. (With no arg, default is True.)
Semantics:	With input True, turns debug mode on. When LOGO is in debug mode, LOGO will generate a Suspend when an error occurs. While execution is suspended (reached this way or by using the :Suspend interrupt character or Suspend explicitly), the user may execute other instructions and then resume execution by using the Continue command. If input is False, exits debug mode.
Errors:	Boolean expected, non-Boolean found.

7.8.3.3 Trace

Command

Syntax:	Trace Specifier
Inputs:	Specifier must be Case 1. a quoted word which names a procedure. Case 2. a flat list -- interpreted as a list of procedure names Case 3. a workspace descriptor list (see 81)
Semantics:	Causes LOGO to begin tracing the procedures and/or names specified in Arg. For traced procedures, causes the trace trap to be sprung on entry to the procedure and when the procedure terminates. For traced names, causes the trace trap to be sprung when the name's thing is accessed or changed (i.e., when the name is bound to a new value). TrapObject contains information indicating the circumstances in which the TraceTrap was sprung. See Untrace, page 102.
Errors:	List argument in wrong format. Non-existent object. Object protection violated.

7.8.3.4 Untrace Command

Syntax: Untrace Specifier

Inputs: Specifier must be

Case 1. a quoted word which names a procedure.

Case 2. a flat list -- interpreted as a list of procedure names

Case 3. a workspace descriptor list (see 81)

Semantics: Turns off tracing. See Trace, page 101.

7.8.3.5 Step Command

Syntax: Step ProcedureSpecifier

Inputs: ProcedureSpecifier must be

Case 1. a quoted word which names a procedure.

Case 2. a flat list -- interpreted as a list of procedure names

Case 3. a workspace descriptor list (see page 81) with procedure names only.

Semantics: Allows the user to monitor the execution of the specified procedure(s) by

- printing the inputs to the procedure
- printing each procedure line and allowing the user to accept or postpone execution of that line. (User types :EOL to run the line. User types text followed by :EOL to run the text; Step then reprints the line for approval or postponement.)
- printing the procedure's output (if any) when it terminates

Execution is interrupted and control is returned to the user without losing the execution environment after every line.

Errors: List argument in wrong format.
Non-existent object.
Object protection violated.

Note: Procedures that have no text (see NoText) cannot be stepped.

7.8.3.6 Suspend

Command

Inputs: None.

Semantics: Execution is interrupted and control is returned to the user without losing the execution environment in which the suspend occurred. Suspends may be trapped when they occur.

7.8.3.7 Continue

Command

Inputs: None.

Semantics: Execution is resumed from a suspension (due to Suspend, the :Suspend interrupt character, or a trap while in debug mode). Continue has no function if executed within a procedure or when no procedure is suspended.

7.9 Input/Output

We call the creation of text forms (external print-representations) of LOGO structures "transcription." To suit the various purposes for which transcription is intended, LOGO provides three transcription forms.

7.9.1 Transcription forms

1. Literal Transcription: Every character is transmitted as itself, with no exceptions, and no extra characters are ever interpolated for formatting or other purposes. Literal transcription is performed by the WriteChar primitive.

2. PrintForm Transcription: Identical with Literal transcription with four exceptions:
   - The # character is transcribed as a :Space.
   - Elements of a list are transcribed with :Space characters separating them.
   - Sublists are transcribed surrounded by square brackets.
   - Care is taken to insert extra formatting characters where necessary to preserve the integrity of words. Thus, when a word to be printed would move the cursor beyond the end of the print line, a carriage return is inserted. Words too long to print on a line are broken at the end of the terminal width with an inserted carriage return.

   This transcription produces the print form of words and lists displayed by Print, whence the name. Lists are printed without outer brackets; words are printed without prefixed double quotes. Characters that have print forms (i.e., most ASCII characters other than control characters and delete) print as themselves, except that "#" is printed as space (as a formatting convenience). Characters without print forms are also transcribed as themselves (and may or may not affect the behavior of the terminal or other device to which they are sent).

   The primitives that do PrintForm transcription are:

   Print
   Type
   LongPrint
   LongType
   Write
   WriteForm

3. Idempotent Transcription: Idempotent transcription is used to create a print-form that will reproduce the original structure if read in again with any of the structured reading routines in LOGO (Read, Request, ...). Most characters (all alphanumerics, some punctuation, and some other characters) are transcribed as themselves in the idempotent form. But certain characters would not read in correctly.

    - Characters such as :Space, "[" and "]" must be expressed differently when they are part of words than when they function as delimiters of a list structure if idempotency is to be preserved. These characters are "protected" by a super-quote character, the at-sign ("@"). On input, the structured read routines treat any non-control character preceded by "@" as literal and not a delimiter. Note that this same super-quote mechanism permits typing of these special characters within words in programs typed from the terminal, if desired. For idempotent transcription, the "#" formatting character is represented by itself; no substitution or super-quote protection is required.

    - Lists are transcribed surrounded by square brackets.

    - Characters that lack a print form (e.g., most control characters) are represented as their character code surrounded by the up-arrow character "^". Thus <Bell> is represented as ^7^ in ASCII systems.

    Primitives that do idempotent transcription are:

    FullPrint FullWrite FullWriteForm

7.9.2 Notes on the transcription of special characters

The internal storage of words and lists places no restriction on the content of words (and thus words in lists). The super-quote and up-arrow forms simply preserve the clarity and integrity of data when it is being entered or transcribed.

In addition to the need for such protection for control characters, and non-syntactic spaces or brackets in certain I/O situations, some characters which may normally be elements of words in data need protection when they might be taken to have syntactic significance by the parser. Parentheses, braces, the comma, the double-quote, the colon, the semi-colon and infix operators, for example, act as word delimiters when parsed, but have no special significance inside a list. If they are to appear in command lines without their special significance, they must be protected from the parser by using the @.

Other characters have special significance for the parser wherever they are encountered. Square-brackets and spaces, for example, are always taken as word delimiters unless they are super-quoted with the @. The at-sign, in turn, is taken as a super-quote for the following character and is not represented in the data unless it is super-quoted. Thus, for @, use @@. Characters which require super-quoting on input (and receive it in Idempotent transcription output) if they are to be considered part of the data rather than word separators or format information include: **:Tab**, **:LF**, **:EOL**, **:Space**, **:Escape**, and @. Literal and PrintForm transcription of these characters gives the characters themselves under all circumstances.

Characters which normally have no print form at all (e.g., **:Delete**, **:WordDelete**, **:DeleteRegion**, **:UntypePhysicalLine**, **:GetKilledText**, etc.) but rather have special functions apart from the stream of characters being typed are, in Idempotent transcription, are transcribed using their character code representation (e.g., ^127^ for <Delete> in ASCII systems). In PrintForm and Literal transcription, they are transcribed as themselves. To input these characters from the terminal, they may be typed as their character codes surrounded by the character "^".

Transcription Examples

Stored data and Literal transc. form	Comments	Transcription Mode	
		PrintForm	Idempotent
AbC	word	AbC	AbC
[AbC]	5 char word	[AbC]	@[AbC@]
@[	word	@[	@@@[
A#B	word	A<space>B	A#B
A<space>B	3 char word	A<space>B	A@<space>B
A<bell>C	3-char word	A<bell>C	A^7^C
[A B C]	3 elt. list	A<sp>B<sp>C	[A<sp>B<sp>C]
[A<sp>B C]	2 elt. list	A<sp>B<sp>C	[A@<sp>B<sp>C]

7.9.3 Terminal-oriented I/O

7.9.3.1 Request Rq

Inputs:	None.
Output:	A list
Semantics:	Pauses to allow user to type input at the terminal. When the user types carriage return, Request returns the input line as a list, converting spaces to element separators, converting open and close square brackets to open and close sublist delimiters, converting character codes surrounded by ^'s to the specified characters, and eliminating redundant separators (multiple separators between elements, separators at the beginning or end of the list, and separators immediately preceded or followed by sublist delimiters). See section 7.9 on idempotency, page 104. If the terminal's cursor or printhead is at the beginning of a line, the user is prompted with a star (*). Otherwise, no prompt is produced.
Synonym:	Rq
Note:	The effect of Request is the same as RequestForm EditString [] []

7.9.3.2 Ask

Syntax:	Ask TypingTime
Inputs:	Positive integer (representing seconds)
Output:	A list
Semantics:	Same as Request except that it terminates and returns Empty if carriage return has not been typed within TypingTime seconds.
Errors:	Number expected, non-numeric object found.

7.9.3.3 EditString EdS

Syntax:	EditString OutputBuffer InputBuffer
Inputs:	Two objects
Output:	A word.
Semantics:	EditString provides special control characters with which the user can edit a line of text and return that edited line to another procedure. EditString makes use of two text areas referred to here as its input buffer and output buffer and initialized with InputBuffer and OutputBuffer, respectively, either or both of which may be empty. See sections 7.13.2, 7.13.3, 7.13.4, and 7.13.5, beginning on page 130 for more description of individual editing characters.

As with other standard terminal input (to monitor at top level or to request), deletion of material currently being typed is possible using :Delete, :WordDelete, :DeleteRegion, :UntypePhysicalLine, and :BackDeleteLogicalLine. All deleted material (except text deleted with a single :Delete not adjacent to another deletion) is stored in a separate buffer called :#Kill#Buffer from which it can be retrieved by :GetKilledText. :GetKilledText appends the contents of :#Kill#Buffer to the end of the output buffer. Consecutive deletes (contiguous segments of deleted text) are appended appropriately to :#Kill#Buffer so that they can be retrieved as a single string. Non-consecutive deletes (except for single :Delete not adjacent to another deletion) replace the contents of :#Kill#Buffer. Single non-contiguous :Delete does not clear :#Kill#Buffer and the character it deletes can not be retrieved with :GetKilledText.

Semantics: (Continued from previous page)
EditString also provides access to the input string specified by InputBuffer. The user can edit this string using the special editing control characters, :ForwardCharacter, :NextWord, :SearchForward, and :EndLogicalLine. These four characters all copy text (character, word, region and logical line, respectively) from the beginning of the input buffer and append it to the end of the output buffer. :Delete, :WordDelete, :DeleteRegion, :UntypePhysicalLine, and :BackDeleteLogicalLine all delete text (character, word, region, physical line and logical line, respectively) from the end of the output buffer (but do not alter the input buffer). New text may be appended at the end of the output buffer merely by typing it. :EOL causes EditString to return the contents of its output buffer in its current state (i.e., without appending remainder of its input buffer). To retain all formatting information, these contents are returned as a word (i.e., in FullWriteForm) which may be converted to a list (with RequestForm) if the user desires to do so.

Text initializing the input buffer of EditString is not typed out for the user unless he requests it with the :RetypeLine character. Text inserted into the output buffer in any way (initialized, newly typed, retrieved from deletions, or copied from the input buffer) is always typed for the user. Deletions from the output buffer are surrounded by \ characters.

Errors: None.
Synonym: EdS

7.9.3.4 Print P Command

Syntax: Print TextString
Inputs: Any object
Semantics: Prints the input object followed by a carriage return. Outermost square brackets and prefixed double-quotes (indicating object type) are not shown by Print. Object delimiters within lists and the sharp-sign (#) within words are printed as spaces. Print displays the PrintForm transcription of its input (see section 7.9, on transcription forms).
Greedy form: LongPrint
Greedy Abbrev: LP

7.9.3.5 FullPrint

Command

Syntax: FullPrint Object

Inputs: Any object.

Semantics: FullPrint displays the Idempotent transcription of its input (see section 7.9, page 104 on idempotency and section 7.13, page 130 on special characters). Thus, FullPrint shows all printing characters exactly and adds the surrounding square brackets for lists. In addition, FullPrint prefixes many special characters with the quoting @. The resulting string, if typed to Request, would generate the original input data structure.

Errors: None.

7.9.3.6 Type

Command

Syntax: Type TextString

Inputs: A single word or list

Semantics: Types the input object at the terminal without a following :EndOfLine character. As with Print, outermost square brackets and prefixed double-quotes (indicating object type) are not shown by Print. Object delimiters within lists and the sharp-sign (#) within words are printed as spaces. See section 7.9, on transcription forms.

Greedy form: LongType

Greedy Abbrev: LType

7.9.4 File-oriented I/O: Reading commands

7.9.4.1 Get

Command

Syntax:	Get FileName OptionalDescriptorList
Inputs:	A word and an optional descriptor list. See section 7.6, page 81.
Semantics:	Adds procedures, names and or arrays specified by the workspace descriptor list (or entire workspace in specified file, if there is no argument following the file name) to user´s workspace. Does not read in procedures from the file if a procedure with the same name is already in the workspace. Notifies the user of this conflict if Get is executed directly by the user at the terminal but does not print a warning if the Get was executed from within another procedure.
Errors:	Non-existent object. Object protection violated. File not accessible. Bad file format.

7.9.4.2 Read

Syntax:	Read FileName
Inputs:	Word
Output:	Object
Semantics:	Reads the next object from the specified file and returns it.
Errors:	Word expected, list found. File not open. Incorrect access to an open file. Ill-formed list object encountered. Unexpected I/O Access Error

7.9.4.3 ReadChar Rch

Syntax: ReadChar FileName
Inputs: Word
Output: Single character word
Semantics: Reads the next character from the specified file and returns it as a word.
Errors: Word expected, list found.
File not open.
Incorrect access to an open file.
Unexpected I/O Access Error

7.9.4.4 ReadLine

Syntax: ReadLine FileName
Inputs: Word
Output: Word
Semantics: Reads the next line of text from the specified file and returns it as a word. Line delimiters are the beginning of file, the end of file, and the :EndOfLine character.
Errors: Word expected, list found.
File not open.
Incorrect access to an open file.
Unexpected I/O Access Error

7.9.4.5 Input?

Syntax: Input? FileName
Inputs: Word naming a file opened in Interactive mode.
Output: A Boolean value.
Semantics: Returns True if there are characters waiting to be read from the input buffer of the specified interactive file, False otherwise.
Errors: Word expected, list found.
File not open.
Incorrect access to an open file.
Unexpected I/O Access Error

7.9.5 File-oriented I/O: Writing commands

7.9.5.1 Save

Command

Syntax: Save FileName OptionalDescriptorList

Inputs: A word and an optional descriptor list.

Semantics: Saves procedures, names and or arrays specified by the workspace descriptor list (or entire workspace , if there is no argument following the file name) on specified file. If file with the same name already exists, asks for explicit confirmation and deletes old file if confirmed.

Errors: Object protection violated.
File not accessible.

7.9.5.2 Write

Command

Syntax: Write FileName TextString

Inputs: Two words.

Semantics: Prints the text string to the specified file and follows it with EndOfLine character. Outermost square brackets and prefixed double-quotes (indicating object type) are not written by Write. Object delimiters within lists and the sharp-sign (#) within words are written as spaces, and sublist delimiters are written as brackets. See section 7.9, on transcription forms.

Errors: Word expected, list found.
File not open.
Incorrect access to an open file.

7.9.5.3 WriteChar

Command

Syntax: WriteChar FileName CharacterString

Inputs: Two words

Semantics: Writes CharacterString to specified file. No padding, separations or other alterations to the data occur.

Errors: Word expected, list found.
File not open.
Incorrect access to an open file.

7.9.5.4 WriteLine

Command

Syntax:	WriteLine FileName String
Inputs:	Two words
Semantics:	Writes the specified text string to the specified file and appends the :EndOfLine character.
Errors:	Word expected, list found. File not open. Incorrect access to an open file.

7.9.5.5 FullWrite

Command

Syntax:	FullWrite FileName Object1 Object2 ... ObjectN
Inputs:	A word and one or more objects.
Semantics:	Same as FullPrint, except that printing is done to the specified file (which must be open).
Errors:	Word expected, list found. File not open. Incorrect access to an open file.

7.9.6 File-oriented I/O: General manipulation

7.9.6.1 Open

Command

Syntax:	Open FileName AccessType
Inputs:	Two words.
Semantics:	Named file is opened with specified access type. (See AccessType, page 116.) If AccessType is "Write, "Append or "Interactive, non-existent files are created and opened. Names of all files opened by Open are put on open files list, returned by All Files.
Errors:	Word expected, list found. Non-existent object. Object protection violated. File not accessible. File already open. Unknown keyword in this context Can't create file.

7.9.6.2 Close

Command

Syntax: Close DescriptorList
Inputs: Descriptor list with Files as first element, or
Flat list interpreted as list of file names.
Semantics: Closes specified file or files.
Errors: List expected, word found.
Non-existent object.
Object protection violated.
File not accessible.
File not open.
List argument in wrong format.

7.9.6.3 Rename

Command

Syntax: Rename OldFileName NewFileName
Inputs: Two words.
Semantics: Renames save file from old name to new name.
Errors: Word expected, list found.
Non-existent object.
Object protection violated.
File not accessible.
Can't create file.

7.9.6.4 Delete

Command

Syntax: Delete FileName
Inputs: Word
Semantics: Deletes specified save file. Whether deleted files are recoverable is dependent on implementation.
Errors: Word expected, list found.
Non-existent object.
Object protection violated.
File not accessible.

7.9.7 File-oriented I/O: Properties

7.9.7.1 EndFile? EOF?

Syntax: EndFile? FileName
Inputs: Word
Output: A Boolean value.
Semantics: Returns True if the character pointer of FileName has moved beyond the last character of the file, False otherwise.
Errors: Word expected, list found.
File not open.
Synonym: EOF?

7.9.7.2 FileSize

Syntax: FileSize FileName
Inputs: Word
Output: Integer
Semantics: Returns the number of characters in the file.
Errors: Word expected, list found.
Non-existent object.
Object protection violated.
Note: File need not be open.

7.9.7.3 AccessType

Syntax: AccessType FileName
Inputs: Word
Output: A word, one of: Unopened, Append, Read, Write, Random, Interactive
Semantics: Returns access type for specified file.
Errors: Word expected, list found.
Non-existent object.
Object protection violated.

7.9.7.4 Protection

Semantics: Return implementation dependent file protection information.
Note: Not in LOGO-S version 0.

7.9.7.5 CharPointer

Syntax: CharPointer FileName
Inputs: Word
Output: Non-negative integer.
Semantics: The character position after which new read or write will begin. (Beginning of file is 0; first character is in position 1.)
Errors: Word expected, list found.
Non-existent object.
Object protection violated.
File not open.

7.9.7.6 SetCharPointer Command

Syntax: SetCharPointer FileName AfterChar
Inputs: Word, non-negative integer.
Semantics: Sets the value of CharPointer. (Beginning of file is 0; first character is in position 1.)
Errors: Word expected, list found.
Number expected, non-numeric object found.

7.9.7.7 Column

Syntax: Column FileName
Inputs: Word
Semantics: Returns the value of the column at which reading or writing is about to take place. (Interactive files, like the terminal, only.)

7.10 Properties

LOGO-S provides a mechanism for assigning properties to names and procedures, generally for the sake of protecting these structures in some way, e.g., read protecting or write protecting them. LOGO also makes use of the concept of Privileged Mode, within which all properties may be changed freely, and Normal Mode, within which structures marked with the property Primitive cannot have any of their properties altered. Privileged Mode may be entered by starting up LOGO in a special way or by using a primitive which creates a local privileged environment within itself.

Default value of all properties is False.

7.10.1 Primitives

7.10.1.1 Property?

Syntax:	Property? DescriptorList Property
Inputs:	A descriptor list specifying a single workspace constituent, and a word specifying a property.
Output:	A Boolean value.
Semantics:	Returns True if the specified workspace constituent has the specified property, False otherwise. Valid properties depend on descriptor type. See list of properties below.
Errors:	Word expected, list found. List expected, word found. List argument in wrong format. Non-existent object. Unknown keyword in this context

7.10.1.2 SetProperty

Command

Syntax: SetProperty DescriptorList Property Value

Inputs: A workspace descriptor list, a word specifying a property, and a value which must be a Boolean.

Semantics: Sets specified property to specified value for specified constituent(s). Valid properties depend on descriptor type. See list of properties below.

Errors: List expected, word found.
List argument in wrong format.
Non-existent object.
Word expected, list found.
Unknown keyword in this context
Object protection violated.
Boolean expected, non-Boolean found.

7.10.2 Properties of Procedures, Names and Arrays

7.10.2.1 "Primitive

Semantics: Procedures with this property can not be Edited, Erased, or Saved. Names and arrays with this property cannot be bound or accessed by the user. Structures with this property are not listed by workspace descriptors except when the descriptor is preceded by Primitive. Thus, Show Procedures will not show procedures for whom this property is True, whereas Show Primitive Procedures will display a list of the title lines of all primitives not marked "Invisible. Similarly Show Primitive Names will display all system protected names and their values, unless the names have the additional property "Invisible. Primitive procedures and names can be listed and Shown, but cannot be Erased, Saved or otherwise manipulated. No properties of primitive structures can be modified outside of privilege mode.

7.10.2.2 "Invisible

Semantics: Constituents with this property are not listed by workspace descriptor primitives. Thus, Show Workspace will only show constituents for whom this property is False. It should be noted that, as a consequence, invisible constituents are not Saved unless Save's descriptor list is constructed by the user. Procedures marked both Invisible and Primitive cannot be executed outside of privilege mode.

7.10.2.3 "Traced

Semantics: Constituents with this property will spring the TraceTrap under circumstances described in Trace. See Trace, page 101.

7.10.2.4 "Stepped

Semantics: Procedures with this property execute a single logical line at a time. See Step, page 102.

7.10.2.5 "NoRead

Semantics: The values of names with this property cannot be accessed.

7.10.2.6 "NoChange

Semantics: The values of names with this property cannot be set. Procedures with this property cannot be edited or erased.

7.11 Turtles and other special devices

Note: These primitives and their specifications will be extended in subsequent planned versions. They are not implemented in LOGO-S version 0.

7.11.1 All turtles

7.11.1.1 Back Bk Command

Syntax: Back Distance
Inputs: Number
Semantics: Causes the turtle to move backward Distance units.
Errors: Number expected, non-numeric object found.
Synonym: Bk

7.11.1.2 Forward Fd Command

Syntax: Forward Distance
Inputs: Number
Semantics: Causes the turtle to move forward Distance units.
Errors: Number expected, non-numeric object found.
Synonym: Fd

7.11.1.3 Right Rt Command

Syntax: Right Angle
Inputs: Number
Semantics: Causes turtle to rotate its heading clockwise Angle degrees.
Errors: Number expected, non-numeric object found.
Synonym: Rt

7.11.1.4 Left Lt Command

Syntax: Left Angle
Inputs: Number
Semantics: Causes turtle to rotate its heading counterclockwise Angle degrees.
Errors: Number expected, non-numeric object found.
Synonym: Lt

7.11.1.5 PenUp PU Command

Inputs: None.
Semantics: Changes turtle state so that subsequent moves will not leave traces.

7.11.1.6 PenDown PD Command

Inputs: None.
Semantics: Changes turtle state so that subsequent moves will leave traces.

7.11.1.7 Horn Command

Syntax: Horn Toots
Inputs: Positive integer
Semantics: Causes turtle to beep Toots times.
Errors: Number expected, non-numeric object found.

7.11.2 Restricted turtles only

7.11.2.1 Lamp Command

Inputs: A Boolean value.
Semantics: Controls lamp on floor turtle. When input is On (True), the light is turned on. When input is Off (False), the light is turned off.
Errors: Boolean expected, non-Boolean found.

7.11.2.2 Here

Inputs: None.
Output: List
Semantics: Returns a list containing (in order) the x and y coordinates of the display turtle and its heading in degrees.

7.11.2.3 XCoord

Inputs: None.
Output: Number
Semantics: Returns the x coordinate of the display turtle.

7.11.2.4 YCoord

Inputs: None.
Output: Number
Semantics: Returns the y coordinate of the display turtle.

7.11.2.5 Heading

Inputs: None.
Output: Number
Semantics: Returns the heading of the display turtle.

7.11.2.6 SetTurtle

Command

Syntax: SetTurtle Coordinates
Inputs: List
Semantics: Sets the display turtle position and heading as specified in Coordinates, a list containing (in order) the x and y coordinates of the turtle and its heading in degrees.
Errors: List expected, word found.
Number expected, non-numeric object found.
List argument in wrong format.
Argument out of range.

7.11.2.7 SetX **Command**

Syntax:	SetX XCoord
Inputs:	Number
Semantics:	Sets the display turtle´s x-coordinate as specified without altering the y-coordinate or heading.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.11.2.8 SetY **Command**

Syntax:	SetY YCoord
Inputs:	Number
Semantics:	Sets the display turtle´s y-coordinate as specified without altering the x-coordinate or heading.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.11.2.9 SetXY **Command**

Syntax:	SetXY XCoord YCoord
Inputs:	Two numbers
Semantics:	Sets the display turtle´s x-coordinate and y-coordinate as specified without altering the heading.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.11.2.10 SetHeading **Command**

Syntax:	SetHeading Heading
Inputs:	Number
Semantics:	Sets the display turtle´s heading as specified without altering the x-coordinate or y-coordinate.
Errors:	Number expected, non-numeric object found. Argument out of range.

7.11.2.11 ClearScreen

Command

Inputs: None.

Semantics: Clears the display screen and sets display turtle at "home," i.e., at coordinates (0, 0) facing "north."

7.11.2.12 Home

Command

Inputs: None.

Semantics: Sets display turtle at "home," i.e., at coordinates (0, 0) facing "north." Equivalent to SetTurtle [0 0 0].

7.11.3 Turtle senses

7.11.3.1 TouchLeft?

Inputs: None.

Output: A Boolean value.

Semantics: Returns True if left touch-sensor of turtle is currently stimulated, False otherwise.

7.11.3.2 TouchRight?

Inputs: None.

Output: A Boolean value.

Semantics: Returns True if right touch-sensor of turtle is currently stimulated, False otherwise.

7.11.3.3 TouchForward?

Inputs: None.

Output: A Boolean value.

Semantics: Returns True if front touch-sensor of turtle is currently stimulated, False otherwise.

7.11.3.4 TouchBack?

Inputs:	None.
Output:	A Boolean value.
Semantics:	Returns True if rear touch-sensor of turtle is currently stimulated, False otherwise.

7.11.4 Other devices

General specifications for extended graphics systems, including the use of color, shading, texture or movement and the generation of hard copy have not yet been developed. Similarly, the appropriate primitives for music or speech synthesis and for input devices other than the keyboard are not defined in Version 0. In many cases, the design and capability to the hardware has not been standardized and the technology is still advancing rapidly. Specification will be developed for later versions of LOGO-S.

7.12 Miscellaneous

7.12.1 Top level control

7.12.1.1 Hello Command

Inputs: None.

Semantics: Provides for user identification to LOGO and initialization of user's environment.

Note: Exact specifications are implementation dependent.

7.12.1.2 Goodbye Bye Command

Inputs: None.

Semantics: Exits LOGO gracefully. A backup copy of the workspace is saved under some system- and, perhaps, user-dependent default name. May ask questions of the user.

Synonym: Bye

7.12.1.3 #Monitor Command

Inputs: ?

Semantics: Runs the show.

7.12.1.4 LastLine LL

Inputs: None.

Semantics: Initializes EditString with :LastLine and executes resulting line after EditString is terminated with a carriage return. Allows re-execution of command line previously typed at top level, with or without modification.

Synonym: LL

7.12.1.5 :LastLine

Contains: Last line typed at terminal at top level.
Note: Lines typed to Request from within procedures are not returned by LastLine.

7.12.1.6 Wait

Command

Syntax: Wait Time
Inputs: Number
Semantics: Causes a pause of Time nths of a second (n depending on implementations) before continuing.

7.12.1.7 Time

Inputs: None.
Output: List
Semantics: Returns current time as list of integers, [hour minute second].

7.12.1.8 Date

Inputs: None.
Output: List
Semantics: Returns current date as list of integers, [year month day].

7.12.1.9 Version

Inputs: None.
Output: List
Semantics: Returns current version number.

7.12.2 Machine level primitives

7.12.2.1 #Peek

Syntax:	#Peek MemoryLocation
Inputs:	Positive integer
Output:	Integer
Semantics:	Returns value in specified memory location.
Errors:	Argument out of range.

7.12.2.2 #Poke

Command

Syntax:	#Poke Value MemoryLocation
Inputs:	Two numbers.
Semantics:	Inserts value into specified memory location.
Errors:	Argument out of range.

7.12.2.3 #Call

Semantics:	Generates call to programs outside of LOGO.
Errors:	Not yet defined.
Note:	Not in LOGO-S version 0.

7.13 Special Characters

Because of the different characteristics of the various machines and operating systems within which the Portable LOGO System is intended to work, we cannot always specify which characters are to be used for which special functions. Some characters are universal; these will be indicated in the sections below. Other characters may differ from implementation to implementation; we will refer to them only by a system name (e.g., :Tab). To maximize consistency among different implementations, the "Contains" section will contain a suggestion for characters to use on ASCII systems.

Each implementor is responsible for defining a character set with the functionalities described in this section.

7.13.1 Characters which interrupt execution

7.13.1.1 :Null

Contains:	Null character
Semantics:	Interrupt character which generates a suspend. Generated by the <Break> key on many terminals, and possibly by other keys (e.g., Control-space and control-@) as well.

7.13.1.2 :Suspend

Contains:	Control Z
Semantics:	"Panic stop." Terminate execution of current procedure.

7.13.2 Editing Characters -- Deleting

Deleted text material (except text deleted with a single :Delete not adjacent to another deletion) is saved in :#Kill#Buffer from which it may be retrieved (until the contents of :#Kill#Buffer are reset) or explicitly preserved in :#Preserve#Buffer for later use. Consecutive deletions are stored together (as a single word) in :#Kill#Buffer and are retrieved whole. Insertions or moves (actions which add to the output buffer) cause subsequent deletions not to be appended to previous contents of :#Kill#Buffer, but rather to replace the previous contents. For retrieval of deleted text, see section 7.13.3 below. See also EditString, page 108.

7.13.2.1 :Delete

Contains:	Delete or rub-out character
Semantics:	Delete last character of EditString's output buffer and append to :#Kill#Buffer. If doing so requires clearing :#Kill#Buffer, wait for another consecutive deletion before inserting killed material. Beeps but does not cause an error if there is nothing to delete.

7.13.2.2 :WordDelete

Contains:	Control W
Semantics:	Delete last word of EditString's output buffer and add to :#Kill#Buffer. Word delimiters are only LineFeed, Tab, :Space, :FormFeed and :CarriageReturn except when they are embedded in words with the super-quoting @ (see @). Beeps but does not cause an error if there is nothing to delete.

138

7.13.2.3 :DeleteRegion

Contains: Control D

Semantics: Delete characters from the end of EditString´s output buffer until first occurrence of character immediately following the :DeleteRegion character and add deleted text to :#Kill#Buffer.

If the character immediately following the :DeleteRegion character is the :DeleteRegion character, itself, the most recent search is repeated (i.e., EditString deletes through the next occurrence of the character last deleted back through by :DeleteRegion).

If the :DeleteRegion character is followed directly by any other editing command character (i.e., :Delete, :WordDelete, :UntypePhysicalLine, :BackDeleteLogicalLine, :ForwardCharacter, :NextWord, :EndLogicalLine, :SearchForward, :GetKilledText, PreserveKilledText, YankPreservedText, ViewPreservedText, or RetypeLine and specifically not including the formatting characters), the delete is cancelled, EditString beeps and no action is taken. To delete backward through one of these characters, one must prefix the sought character with @. To search for @, use @@.

If the character to be deleted back through cannot be found, EditString beeps and no characters are transferred (i.e., the cursor does not move).

7.13.2.4 :UntypePhysicalLine

Contains: Control U

Semantics: Delete to beginning of physical line (i.e., backward to but not including last :LF in EditString´s output buffer) and add deleted text to :#Kill#Buffer. Beeps but does not cause an error if there is nothing to delete.

7.13.2.5 :BackDeleteLogicalLine

Contains: Control B

Semantics: Delete text (in EditString's output buffer) backward to beginning of logical line and add deleted text to :#Kill#Buffer. Beeps but does not cause an error if there is nothing to delete.

7.13.3 Editing Characters -- Preserving/Retrieving/Viewing killed text

Text initializing the input buffer of EditString is not typed out for the user unless he requests it with the :RetypeLine character, described in this section. Text inserted into the output buffer in any way (initialized, newly typed, retrieved from deletions, or copied from the input buffer) is always typed for the user. Thus, GetKilledText and YankPreservedText type out what they retrieve.

7.13.3.1 :GetKilledText

Contains: Control G

Semantics: Get back most recently killed text from :#Kill#Buffer and append to end of the output buffer of EditString. Contents of :#Kill#Buffer are not altered.

7.13.3.2 :PreserveKilledText

Contains: Control P

Semantics: Explicitly preserve killed text by copying from :#Kill#Buffer into :#Preserve#Buffer. Contents of :#Kill#Buffer are not altered. Preserved text may later be yanked back with :YankPreservedText for appending to end of the output buffer of EditString.

7.13.3.3 :YankPreservedText

Contains: Control Y

Semantics: Yank back text from :#Preserve#Buffer (preserved by the :PreserveKilledText character) and append to end of the output buffer of EditString. Contents of :#Preserve#Buffer are not altered.

7.13.3.4 :ViewPreservedText

Contains: Control V

Semantics: View text in :#Preserve#Buffer. A line-feed is typed, the preserved text is typed, a carriage-return line-feed is typed and the last physical line of the output buffer is retyped. Neither :#Preserve#Buffer nor the output buffer are altered.

7.13.3.5 :RetypeLine

Contains: Control R

Semantics: Retype (separately) the input buffer and output buffer of EditString.

7.13.4 "White space" characters

Note: All white-space characters are word delimiters. Some have additional properties affecting the formatting or termination of an input line.

7.13.4.1 :Space

Semantics: Word delimiter.

7.13.4.2 :Tab

Contains:	Control I
Semantics:	Space to nearest multiple of 8 on current line.

7.13.4.3 :LF

Contains:	LineFeed
Semantics:	Provide new physical line within logical line.

7.13.4.4 :EOL

Contains:	Carriage return
Semantics:	Terminate and execute command line. Sends a bell character (within Request or EditString) when a list or sub-list remains unclosed.

7.13.5 Editing Characters -- Copy/Keeping input text

Text initializing the input buffer of EditString is not typed out for the user unless he requests it with the :RetypeLine character. Text inserted into the output buffer in any way (initialized, newly typed, retrieved from deletions, or copied from the input buffer) is always typed for the user. Thus all commands in this section type out what they transfer.

7.13.5.1 :ForwardCharacter

Contains:	Control F
Semantics:	Copy character from beginning of EditString input buffer and append it to end of the output buffer of EditString. Beeps but does not cause an error if there is nothing to copy.

7.13.5.2 :NextWord

Contains: Control N

Semantics: Copy word from beginning of input buffer of EditString and append it to end of the output buffer of EditString. Word delimiters are only LineFeed, Tab, :Space, :FormFeed and :EndoflinE except when they are embedded in words with the super-quoting @ (see @). Beeps but does not cause an error if there is nothing to copy.

7.13.5.3 :SearchForward

Contains: Control S

Semantics: Copy region from beginning of input buffer of EditString to first occurrence of character immediately following the :SearchForward character, and append copied text to end of the output buffer of EditString. If the character immediately following the :SearchForward character is the :SearchForward character, itself, the most recent search is repeated (i.e., EditString copies through the next occurrence of the character last searched for).

If the :SearchForward character is followed directly by any other editing command character (i.e., :Delete, :WordDelete, :DeleteRegion, :UntypePhysicalLine, :BackDeleteLogicalLine, :ForwardCharacter, :NextWord, :EndLogicalLine, :GetKilledText, PreserveKilledText, YankPreservedText, ViewPreservedText, or RetypeLine and specifically not including the formatting characters), the search is cancelled, EditString beeps and no action is taken. To search for one of these characters, one must prefix the sought character with @. To search for @, use @@.

If the character searched for cannot be found, EditString beeps and no characters are transferred (i.e., the cursor does not move). Search for a :EndoflinE copies to the end of the logical line unless there is a :EndoflinE character embedded in the text of the line.

7.13.5.4 :EndLogicalLine

Contains: Control E

Semantics: Copy entire input buffer and append copied text to end of the output buffer of EditString. Beeps but does not cause an error if there is nothing to copy.

7.13.6 Characters without interrupt status

7.13.6.1 :

Semantics: Prepended decoration ("dots") indicating that following literal is to be treated as an evaluated name. "Dots" must be followed immediately by a name or by another "dots." See also Thing.

7.13.6.2 "

Semantics: Quote-mark prepended to literals on type-in.

7.13.6.3 @

Semantics: Super-quote character for protecting other specialized characters and causing them to be interpreted as if they were alphabetic (i.e., with no special meaning).

7.13.6.4 $

Semantics: Character that Prints as a space for formatting purposes.

7.13.7 Delimiters

7.13.7.1 [

Semantics: Opening list (sub-list) delimiter.

7.13.7.2]

Semantics: Closing list (sub-list) delimiter. Simulated (within Request or EditString) by :EOL when a list or sub-list remains unclosed.

7.13.7.3 (

Semantics: Opening delimiter indicating the scope of a greedy or the precedence of operations in an arithmetic statement.

7.13.7.4)

Semantics: Closing delimiter indicating the scope of a greedy or the precedence of operations in an arithmetic statement.

7.13.7.5 {

Semantics: Opening delimiter for groups of commands that are scoped together.

7.13.7.6 } }

Semantics: Closing delimiter for groups of commands that are scoped together.

7.13.7.7 ^

Semantics: Character code delimiter.

7.13.7.8 !

Semantics: Delimiter separating comments from executable code in procedure lines. Mark both beginning and end of commented text.

7.13.7.9 ,

Semantics: Conditional delimiter between Boolean condition and statement sequence in conditional statements. Synonym for Do and Then.

7.13.7.10 ;

Semantics: Delimiter between commands that reside on the same logical line.

7.13.7.11 :Escape

Contains: Escape (altmode) character
Semantics: Special function character.
Note: Not in LOGO-S version 0.

7.14 Error types

7.14.1 Errors occurring primarily during parsing

Procedure called, not defined.

Wrong number of arguments.

Misplaced syntactic marker:
    Expecting Then, Do, Else or comma, but found something else.
    Found Then, Do, Else or comma when not expecting it.

7.14.2 Errors occurring during evaluation

Word expected, list found.

List expected, word found.
Single character word expected, something else found.

Number expected, non-numeric object found.

Non-numeric word expected, number found.

List argument in wrong format.

Boolean expected, non-Boolean found.

Unknown keyword in this context
    Unknown trap type.
    Non-existent open file mode.
    Non-existent property.

Division by zero.

Argument out of range.

Command expected, operation found.
    Missing argument. Also occurs if a command is encountered within Value or UnwindWith.

Operation expected, command found.
    Unclaimed argument left over after all procedures have been evaluated. Also occurs if a value is encountered

within Run or Unwind.

Ill-formed list object encountered.
Conversion of a character string to list form requires that brackets be balanced. This occurs when **Read** (or **RequestForm**) is constructing a list object and encounters an **:EndOfFile** character (or end of the data object) before the list terminator.

Non-existent object.

Undefined name.

Procedure already exists

Object protection violated.
Procedure cannot be trapped.

Executed while not in procedure.

Trap not in progress.

Can't create file.

File not accessible.
E.g., open by someone else.

File not open.

File already open.

Incorrect access to an open file.

Bad file format.
For Get.

Unexpected I/O Access Error
File clobbered between Open and Read.

PART 3

A GUIDE TO IMPLEMENTATION

8. AN OVERVIEW OF THE PORTABLE LOGO SYSTEM

This document is a description of the BBN Portable LOGO system. It is intended for the LOGO implementor who wishes to use the Portable LOGO system in developing an interpreter for LOGO-S on a specific machine. Included are the specifications of the tasks required for implementation, and the description, at a detailed level, of the modules which comprise the system.

The BBN Portable LOGO system is designed to allow the implementation of an interpreter for LOGO-S on a variety of machines with a minimum of effort. (The LOGO-S language is specified in Part II). In this chapter we describe the overall structure of Portable LOGO, with respect to the tasks required for implementation.

8.1 Components of Portable LOGO

BBN Portable LOGO has four components. The first is the LIL translator. LIL (LOGO Implementation Language) is the programming language in which much of portable LOGO is coded. The translator is used to build a local implementation of the LOGO machine.

The LOGO machine, written in LIL, obtains a LOGO-S command line or procedure which is parsed into an internal form, and then evaluated. Most of the primitive commands and operations defined in LOGO-S are implemented as subroutines within the LOGO machine, invoked during the evaluation process.

THe LOGO machine operates in a host environment, which provides an interface between Portable LOGO and the host machine (usually through the local operating system), and supports the execution of the translated LIL code.

The user of the portable LOGO system sees the LOGO machine through the LOGO-S environment, a collection of LOGO-S procedures. These procedures implement defined LOGO-S primitives, which provide some of the higher-level functions of LOGO-S, such as error reporting and LOGO-S procedure editing and debugging.

The major task of the implementor is to provide the LIL translator and the host environment. The LOGO machine and the LOGO-S environment comprise the portable LOGO code distributed by BBN.

8.2 General organization

The Portable LOGO system, when executing, is composed of three parts: a code area, a constant area, and a heap. The code area is used for storage of the assembled LOGO machine and host interface program; the constant area contains data structures which are used to initialize the system (names of primitive LOGO-S functions, for example), and thus may be overlaid; and the heap contains dynamically-allocated data structures of the LOGO machine (such as LOGO-S procedures and LOGO-S data, including the LOGO-S environment).

8.3 Outline of Implementation Tasks

The following is a broad outline of the tasks required to implement a local version of portable LOGO, with references to those parts of this document which provide greater detail. These tasks need not be done serially. A reasonable strategy is to develop the LIL translator and host interface program in parallel, since one cannot be tested without the other. An implementation of portable LOGO should be an iterative process; the first priority should be to get the system running, and then to fine tune such things as the design of storage allocator and the expansion of LIL macros.

1. Build the host environment (specified in chapter 10), written in a local system-building language of the implementor's choice. This includes:
    a. The defined host interface routines (Section 10.2).

    b. The LOGO machine's dynamic storage allocator (Section 10.1).

    c. Other routines as necessary to support the LIL object code. This will vary with method used to implement LIL (Section 9.1).

2. Build the LIL translator (LIL is specified in Chapter 9). The tasks here can vary greatly, but will usually involve:
    a. Writing a set of macros in the local macro-assembler to assemble LIL code. (Obviously, an implementor should be sure that a macro-assembler of sufficient power is available before starting).

    b. Pre-processing the supplied LIL code into a form acceptable to the local assembler. This might involve writing such a pre-processor from scratch, or making global changes to the source code with a local text editor, or some combination of the two.

152

3. Modifying implementation-dependent constants within the LOGO machine and the LOGO-S environment. This includes such things as buffer sizes, file names, and the character-encoding scheme (the distributed system uses the ASCII character set).

4. Assembling and/or compiling the host interface program and the LOGO machine code, and linking them into a run module, using the local system utilities.

   At this point, one would have a running system. However, an implementor may wish to take the following optional steps:

5. Modifying the LOGO-S environment to take advantage of local hardware, such as special display terminals.

6. Adding the LOGO-S turtle graphics primitives, which are defined but not implemented because they are so hardware dependent. Normally, this would be done by augmenting the LOGO-S environment, but one might add them to the LOGO machine because of hardware or execution speed requirements.

7. Making other local modifications to the LOGO machine to take advantage of special capabilities of the local system.

9. THE LOGO IMPLEMENTATION LANGUAGE

9.1 Introduction

This chapter is a description of LIL, the LOGO Implementation Language, and its use in the coding of the LOGO machine. LIL is a low-level programming language designed to be a convenient implementation tool for the BBN LOGO implementation, and to be easily implementable on many computers. For the purposes of this discussion, it is helpful to assume the existence of a hypothetical LIL machine with an instruction set consisting of a subset of LIL. The LIL description is artificially segmented into a description of the machine instructions and a description of the assembly language. In practice, an implementor may not make this distinction.

LIL may be implemented in a number of ways: a LIL interpreter is conceivable; LIL may be micro-programmed on an appropriate computer (this is of course a variety of interpretation); but most often, the implementation will use a macro-processor to expand each LIL statement into one or more target instructions. Depending on the complexity of a LIL instruction, target instructions may call subroutines within the host environment (this is almost certainly the case with LIL instructions which dynamically allocate storage).

9.2 The LIL machine

The LIL machine is a virtual machine which is designed to be easily implemented on a broad range of target machines. It has a register-oriented instruction repertoire, with a range of addressing modes. All instruction types are tagged. Thus there is, for example, a ´move integer´ (MOVI) and a ´move pointer´ (MOVP) instruction, even though on many machines both LIL instructions would generate the same machine instruction.

The instruction repertoire is highly orthogonal; for example, all combinations of conversion operations are provided, including a few silly ones, such as ´convert string to pointer´ (CVTSP). Naturally, no implementor is expected to provide instructions such as these; however, the existence of the silly instructions makes the instruction repertoire much more regular than most actual machine architectures.

The addressing modes are basically those of the PDP-11, with the exception of auto-increment and decrement. An operand may reside in a register (register mode) or in a memory cell. If the latter is the case, the operand may be held in the instruction itself (immediate mode), directly addressed in the instruction (memory mode), or addressed through a register offset by an assembly-time constant (indexed mode). The following LIL program fragment moves the contents of one memory cell (FOO) to another (GOO), using all four modes:

```
    movi    FOO,r2            ;move the (integer) contents of
                              ; FOO to register 2
    movp    <#,GOO>,r1        ;move the address of the cell GOO
                              ; to register 1
    movi    r2,<@,1>          ;move the contents of register to
                              ; the memory cell whose address
                              ; is in register 1 (GOO)
```

This example is rather long-winded; LIL instructions may be memory-to-memory. The following LIL statement has equivalent effect (except that the contents of the registers are not affected):

```
    movi    FOO,GOO           ;move the (integer) contents of
                              ; FOO to GOO
```

The basic data types of the LIL machine are pointers and integers. Integers are expressed in some signed binary format which is unspecified, but very likely to be twos-complement.

Pointers are unsigned, and are of three types:

1. **Address pointers** reference locations in the assembled LIL code, either data cells or LIL program branch locations. Their values are obtained at assembly time, from program labels within the LIL source code.

2. **Object pointers** reference data aggregates created during the execution of a LIL program.

3. **Fake pointers** are special values, defined at assembly time, which are manipulated by those LIL instructions which take pointers as operands, but do not reference any actual location within the LIL machine. Of special interest to the LIL implementor is the null pointer, denoted by the symbol **NULL** in LIL source code. The range of values of object pointers and fake pointers are guaranteed not to overlap.

No assumption is made about the natural addressing unit of the target machine. Pointers on byte-addressable machines are assumed to be aligned as necessary.

Memory cells may be declared to contain either integers or pointers, but not both, as they may be of different lengths in a particular implementation. Integers must contain at least 16 bits.

General purpose registers (designated Rn) are of sufficient length to contain either integers or pointers. When used as index registers, their contents are treated as pointers. There are 5 such registers, numbered 0 through 4. The following principle is worth noting: The access time of register n is less than or equal to that of register n+1. For this reason, a standard LIL coding convention is that the most frequently accessed operands are in the lowest numbered registers.

In addition, there are a number of registers which are not accessible to the LIL programmer. A stack pointer is used to access the system stack, a program counter records the address of the next instruction to be executed, and there are two error registers.

9.3 LIL Data Objects

LIL instructions reference two types of operands: single cells allocated at assembly time, and, aggregates of bytes or words which are usually allocated dynamically. (A word is defined to be large enough to contain either an integer or a pointer). The latter are referred to as data objects. All LOGO-S data structures are implemented as LIL data objects. LIL data objects are of three varieties--vectors, numbers and strings.

All LIL data objects are stored with a prefix field, which contains at least two pieces of information--a length (exclusive of the prefix), and a type (or flag) field. The latter (with the exception of a few bits) is unused by the storage allocator, and is provided for the portable LOGO system itself. There may be additional implementation-dependent information as well.

References to data objects are made through pointers. A data object pointer always refers to the beginning of the data portion of an object, i.e., directly following the prefix, as in the following diagram:

```
*----------------------------*
*          prefix            *     *--------------*
*----------------------------* <---*object pointer*
*                            *     *--------------*
*          data              *
*                            *
*----------------------------*
```

9.3.1 Vectors

A vector consists of a number of integers followed by pointers to other LIL data objects. The integers are used as, for example, array bounds. Thus a vector has the following organization:

```
*------------------------------*
*           prefix             *
*------------------------------*
*           nints              *
*------------------------------*
*                              *
*          integers            *
*                              *
*------------------------------*
*                              *
*          pointers            *
*                              *
*------------------------------*
```

nints, the first word in a vector, is an integer indicating how many integers (including nints) are present. The length field of a vector prefix counts the total number of words in the vector, including the nints word. A vector word must be of sufficient length to accommodate both integers and pointers.

A vector is marked by the **F$VEC** bit in the type field of the prefix.

9.3.2 Numbers

A number is actually a vector which contains only integers. Thus, numbers are manipulated by the same LIL instructions which operate on vectors. The only distinction made between vectors and numbers are in the interpretation of the first (NINTS) word, which has no special meaning in a number. Numbers have the **F$NUM** bit on in their prefix type fields.

9.3.3 Strings

A string consists (after the prefix) of a sequence of bytes. No assumptions are made about the contents of a string. The **F$STR** bit in the type field marks an object as a string.

Note that, in a string, the length field counts the number of valid bytes. In reality, many machines impose some sort of alignment constraints. Therefore, there may be slack bytes between objects. It is the responsibility of the storage allocator to care for these bytes.

9.3.4 Dynamically-created data objects

Most LIL data objects are stored in a memory area called the heap. This term (originated by the ALGOL 68 group) refers to a storage regime in which objects are directly accessible via external pointers, and in which storage reclamation, involving the relocation of objects within the heap, may take place any time an attempt is made to allocate a new object. All heap data objects, or heapjects, are indeed allocated dynamically, either through the explicit invocation of the host routine $ALLOC (see Section 10.2.2), or implicitly through the execution of a LIL instruction.

Heapjects are referenced under the "one-owner discipline", a scheme to minimize the complexity of designing the storage allocator. Each heapject has exactly one owner, i.e., is referenced by one pointer. (This is discussed further in section 10.1). This has several significant effects upon the LIL machine and the coding of LIL programs.

It should first be noted that placing a new value in a memory cell which previously referenced ("owned") a heapject effectively de-allocates the object, since there is no longer any means of accessing its contents. Therefore, constructs such as:

```
    p := alloc()
```

cannot be used, since the allocator must know to what p pointed before. Instead, a style such as

```
    alloc(&p)
```

(in which "&" means "address of") is employed--the allocator must free what p pointed to, and then set it to reference the newly created object.

Secondly, copying the contents of an "owner" into another cell creates an ambiguous situation in that there are now two pointers to the same heapject, violating one-ownership. Memory cells which reference objects without "owning" them are called transient pointers, and are occasionally necessary. Because of this, LIL provides several 'move pointer' instructions, some of which not only transfer data, but ownership as well. A cell containing a transient pointer which is the source of such an operation will transfer ownership even though it was not itself the owner. The LIL programmer must be on guard for such 'side effects'. Also, the execution of a LIL instruction which creates a new heapject immediately invalidates the contents of all transient pointers. However, a transient pointer may be the source of such an operation, even though its contents afterwards become meaningless.

Finally, accessing the contents of a heapject is somewhat complicated by not being able to maintain pointers to specific bytes (for strings) or words (for vectors) within the object. LIL instructions are provided which read and write the contents of a heapject using integer offsets from the "owning" pointer. Thus, we have written the LOGO machine code with an eye towards minimizing references to the heap. We have assumed that accessing the contents of a data object is a relatively expensive operation.

9.3.5 Pseudo Heap Objects

There are also LIL assembly directives which create data objects in-line (and thus outside the heap) at assembly time. These objects are not restricted by the one-owner discipline, but can be used in all instructions which reference the contents of heapjects; thus, they are referred to as pseudo heap objects.

For example, the LIL code sequence

```
            pfix F$STR,9
    LABEL:  text "a message"
```

directs the assembler to create a pseudo-heap object, marked as a string and with a length of 9 bytes. The assembler will also create the symbol **LABEL** which can be used to point to the string (in immediate address mode).

9.4 Name tables

Name tables are hash tables, represented as LIL vectors of pointers. Each pointer in the hash table references a linked list of entries. This method is relatively storage efficient, in that the hash tables themselves may be quite small, and other space is not used until needed. Furthermore, rehashing is never needed, since the tables can grow without limit. The space for the table entries themselves is preallocated; entries are managed by free lists. It should be noted that, unlike other dynamically allocated LIL objects, the "one owner discipline" does not apply to name table entries.

Although the exact structure of a table entry is implementation-dependent, it has the following components:

- A name field, pointing to a string which is the retrieval key.
- A value field, which is a pointer to a data object, either a string or a vector.
- A flag field, containing an integer of at least 16 bits.
- A link field, pointing to the next entry with the same hash code. The link field in the last entry in a list of entries contains the null pointer.

LIL instructions are defined which read and write the components of a table entry.

The standard LOGO name tables are accessed via hash tables allocated at assembly time. (Dynamically created name tables may be implemented at a later date). Thus, the hash tables are "pseudo-vectors". However, the NINTS word gives the size of the hash table, rather than the number of integers in the vector, which would be 1. (Note that the value of the NINTS word includes itself, i.e., is one greater than the number of actual hash "buckets").

9.5 The LIL Instruction Set

This section contains a summary and specification of the instruction set of the LIL machine. Instructions are shown in a format approximating that of the assembly language: an instruction name, followed by a list of operands. Instructions which take optional operands show their optional parts enclosed in square brackets. Generally, an instruction name has a suffix which denotes the type of its operands: I for integers, P for pointers, S for pointers to strings, VEC for pointers to vectors. The lower-case letters ´m´ or ´n´ mean an instruction has both an integer (I) and a pointer (P) form.

Operands are denoted as one of: src, an operand of any mode, i.e., a source operand; dst, any operand which is not in immediate mode, i.e., a destination operand; loc, an operand in main memory (not a register), i.e., a program branch.

9.5.1 LIL Instruction Errors

Many LIL instructions provide an optional error operand. If this is specified, a detection of an error in the execution of the operation will cause an ´error jump´. This causes an error code and the current machine program counter to be placed in two error registers, and control transferred to the location specified by the error operand. The GETERR instruction provides a means of accessing these registers.

The following is a list of error codes which can be inserted into the error code register by LIL instructions, denoted by the symbols used to reference them in the LOGO machine. (The actual values of these symbols are specified in module ´ERRORS´ of the LOGO machine code). These do not represent all the conceivable errors, but rather those errors for which it is not more convenient to insert specific checks into the code; an implementor during the debugging process may wish to define additional errors and add error operands to the distributed code temporarily:

E.SOE	An offset used to reference a string was incorrect.

E.VOE	An offset used to reference a vector was incorrect.
E.ANP	An attempt was made to advance the null pointer (GETNXT only)
E.OVL	An arithmetic instruction caused overflow.
E.DIV	An arithmetic instruction caused a divide check.

It should be noted that in the distributed LOGO machine code the error jump operand is most often omitted. The implementor may want to take advantage of this in optimizing the expansions of the LIL instructions.

9.5.2 LIL Instruction Summary

```
MOVm    src,dst
OMOVP   src,dst
SWAPP   src,dst
ADDm    src,dst [,loc]
SUBm    src,dst [,loc]
MULm    src,dst [,loc]
QUOm    src,dst [,loc]
REMm    src,dst [,loc]
ANDm    src,dst
IORm    src,dst
XORm    src,dst

INCm    dst[, loc]
DECm    dst[, loc]

LADDI   src, dst, loc
LDIVI   src, dst, dst [,loc]
UDIVI   src, dst, dst [,loc]

MOVS    src, dst
CATS    src, src, dst
CMPS    src, src, loc, loc
CMPSU   src, src, loc, loc
SUBSTR  src, dst, [ ,src [ ,src [ ,loc ] ] ]
RPLSTR  src, dst, [ ,src [ ,src [ ,loc ] ] ]
EXTS    src, dst, [ ,src [ ,src [ ,loc ] ] ]
INSS    src, dst, [ ,src [ ,loc ] ]
APPS    src, src, dst, src, [ ,loc ]

SUBVEC  src, dst, [ ,src [ ,src [ ,loc ] ] ]
```

```
RPLVEC  src, dst, [ ,src [ ,src [ ,loc ] ] ]
EXTN    src, dst, [ ,src [ ,src [ ,loc ] ] ]
INSN    src, dst, [ ,src [ ,loc ] ]

CVTmm   src, dst[, loc]
CVTmS   src, dst
CVTSm   src, dst[, loc]
CVTID   dst [,loc]
CVTDI   dst [,loc]

JMP     loc

JMP[N]Lm src, src, loc
      E
      G

JMP[N]ZI src, loc
      P
      M

ENTER   src, src, dst [,loc]
FIND    src, src, dst
REMOVE  src, src [,loc]
GETVAL  src, dst
SETVAL  src, dst
GETFLG  src, dst
SETFLG  src, dst
GETNAM  src, dst
GETNXT  src, dst [,loc]

GETTYP  src, dst
GETLEN  src, dst
SETTYP  src, dst
SETLEN  src, dst

PUSHM   Rn,Rn
PUSH    Rn
PUSHm   src
CALL    loc [ ,loc ]
PCALL   loc [ ,loc ]
RETURN
FAIL
SUCCEED
POPM    Rn,Rn
POP     Rn
POPm    dst

HOST    addr
```

```
        TSTB    src,src,loc
        SETB    src,dst
        CLRB    src,dst

        SETERR  src, loc
        GETERR  dst, dst
```

9.5.3 Basic Double Operand Instructions

MOVm src, dst
Move the source field to the destination field.

SWAPP src,dst
SWAPP is a ´move pointer´ operation which maintains the ´one-owner´ discipline. It examines its operands to see if they are ´heapjects´ (heap objects, either strings or vectors. See Section 9.3.4 for details on the heap and the one-owner discipline).The following actions are performed, in order:

1. If dst points to a ´heapject´, deallocate it. Note that dst might not be the doomed object´s owner, in which case the operation has a side effect. To avoid this, use OMOVP.

2. Move the contents of src to dst.

3. If src points to a ´heapject´,make dst its (the heapject´s) ´owner´ and replace the contents of src with the null pointer.

If the source operand of SWAPP was not an immediate mode operand, its contents are guaranteed not to point into the heap after execution of the instruction.

OMOVP src,dst
OMOVP is intermediate in effect between MOVP, which transers data between source and destination ignoring one-ownership, and SWAPP, which completely preserves one-ownership. In OMOVP, dst becomes the owner of src, but the former contents of dst are not deallocated. This instruction is used most often when transferring ownership to register which formerly contained an integer, or, to a transient pointer.

If the source operand of OMOVP was not an immediate mode operand, its contents are guaranteed not to point into the heap after execution of the instruction.

ADDm src, dst[, loc]
Add the source to the destination. If an overflow occurs, jump to the specified location.

SUBm src, dst[, loc]
Subtract the source from the destination. If an overflow occurs, jump to the specified location.

MULm src, dst[, loc]
Perform an integer multiplication (result has the same precision as the operands). loc is the address of an overflow routine.

QUOm src, dst[, loc]
Compute the quotient of the destination divided by the source. If a divide check occurs, branch to loc.

REMm src, dst[, loc]
Compute the remainder of the destination divided by the source. Note that the remainder is only defined if both operands are positive. If a divide check occurs, branch to loc.

ANDm src, dst
Set the destination to the logical AND of the two operands.

IORm src, dst
Compute the logical inclusive OR of the operands.

XORm src, dst
Compute the logical exclusive OR of the operands.

9.5.4 Basic Single Operand Instructions

INCm dst[, loc]
Increment the operand by an appropriate value (1 for integers; 1 word for data pointers). Jump to loc on overflow.

DECm dst[, loc]
As for INCm, but subtract the appropriate value. Jump to loc on overflow.

9.5.5 Double Length Arithmetic

The following instructions all form double-length results from single-length integer operands. The high and low order words of this result will have the same sign (if both are non-zero).

LADDI src, dst, loc
Add the src to dst, forming a double-length result. Place the low order word in dst. Jump to loc if the high-order word is non-zero.

This instruction differs from ADDI in that the result is well-defined in the case of overflow.

LDIVI src, dst1, dst2 [,loc]
Long divide. The double length integer formed by dst1 (high order) and dst2 (low order) is divided by src, quotient to dst2, remainder (of same sign as quotient) to dst1. Error jump on zero divide or if quotient will not fit in single word (i.e., dst1 must initially be less than src).

UDIVI src, dst1, dst2 [,loc]
Undivide. Double-length product of src & dst1 is added to dst2 to produce double-length result. High order goes to dst2, low order to dst1.

9.5.6 Control flow

JMP loc
Transfer control to loc unconditionally.

JMPZI src, loc
If the source operand is 0, jump to loc; otherwise, continue with the next instruction.

JMPNZI src, loc
Jump to loc if the operand is not 0.

JMPPI src, loc
Jump to loc if the operand is positive (plus).

JMPNPI src, loc
Jump to loc if the operand is not positive.

JMPMI src, loc
Jump to loc if the operand is negative (minus).

JMPNMI src, loc
Jump to loc if the operand is not negative.

JMPLm src, src, loc
If the first operand is less than the second, jump to loc; otherwise continue with the next instruction.

JMPNLm src, src, loc
If the first operand is less than the second, continue with the next instruction; otherwise, jump to loc.

JMPEm src, src, loc
Jump to loc if the operands are equal.

JMPNEm src, src, loc
Jump to loc if the operands are not equal.

JMPGm src, src, loc
Jump to loc if the first operand is greater than the second.

JMPNGm src, src, loc
Jump to loc if the first operand is not greater than the second.

9.5.7 String operations

In the following operations, note that a call to the storage allocator is implied in all operations which create a new string (**MOVS**, **CATS**, etc.). This means that the result of a string operation which has identical source and destination operands is undefined. As noted in the previous discussion of transient pointers, it is permitted to have a transient pointer as source of a copying operation.

The objects created by these operations will not necessarily have the same flags as their sources. Only the string type flag (F$STR) will be set.

MOVS src, dst
Copy the string pointed to by the src and place a pointer to the new string in dst.

CATS src, src, dst
Copy the concatenation of the two strings referenced by the sources and place a pointer to the new string in dst.

CMPS src, src, loc1, loc2
Compare the two strings pointed to by the source designators, with respect to a not defined collating sequence. If the first operand is greater, jump to loc1; if the second is greater,jump to loc2. If the strings are identical,fall through to the next instruction.

CMPSU src, src, loc1, loc2
Identical to CMPS, with the exception that upper & lower case letters are considered equivalent.

SUBSTR src,buf,[offset [,nchar [,loc]]]
Source string is copied into a buffer of nchar words (1 right- justified byte per word) referenced by buf, starting from the offseth byte in src. If nchar is not specified, it is assumed to be 1. If offset is not specified, it is assumed to be 1. The error jump is taken if an attempt is made to copy beyond the end of the string.

RPLSTR buf, dst[,offset [,nchar [,loc]]]
The first nchar words of the source buffer referenced by buf are copied into the destination string dst. If nchar is not specified, it is assumed to be 1. If offset is not specified, it is assumed to be 1. The error jump is taken if an attempt is made to copy beyond the end of the string.

INSS src, dst [, offset [,loc]]
Insert string, similar to RPLSTR. No new storage is allocated. The entire contents of the string referenced by SRC are inserted into the string referenced by dst, starting at the offseth byte (which defaults to 1). The error jump is taken if an attempt is made to insert beyond the end of the destination string.

EXTS src, dst [,offset [,nchars [,loc]]]
A pointer to a newly allocated string is placed in dst, containing a substring of the source string referenced by src, of length nchars and starting at offset (which default to 1). The error jump is taken if an attempt is made to copy beyond the end of the source string.

APPS buf,src,dst,nchar [,loc]
Append the first nchar words, interpreted as right-justified characters, from the buffer referenced by buf, to the string src, and copy into a

newly allocated string to be referenced by dst. The error jump is taken if nchar is negative. If nchar is zero, this operation is equivalent to **MOVS**.

9.5.8 Vector operations

These operations should be the only method used to access the contents of heap vectors. Not only are they guaranteed to preserve the storage allocation philosophy of LOGO-S, but they allow optimal packing of data within vectors and may facilitate implementations on machines with memory management.

SUBVEC src,buf [,offset [,nwords [,loc]]]
Copy the vector specified by src into a buffer specified by buf. Start at the offseth word within src and copy nwords words. Take the error jump if an attempt is made to copy from outside the vector, but not before copying as much as possible. The default values for offset and nwords are both 1. Pointers moved into the buffer are ´transient´, and the referenced object remains the ´property´ of the source vector.

RPLVEC buf,dst [,offset [,nwords [,loc]]]
Same as for **SUBVEC**, but copy from buffer into vector specified by dst. Like **SWAPP**, **RPLVEC** also maintains the ´one-owner´ discipline for any pointers which it moves,i.e., the destination vector becomes the "owner" of the moved pointers and its former contents are deallocated and replaced with null pointers.

9.5.9 Vector operations for numbers only

The following operations do not preserve the one-owner discipline and thus should only be used on numbers, or the integer portions of vectors.

INSN src, dst [, offset [,loc]]
Insert number. No new storage is allocated. The contents of the number referenced by src are inserted into the number referenced by dst, starting at offset (which defaults to 1). The error jump is taken if an attempt is made to insert beyond the end of the destination number.

EXTN src, dst [,offset [,nwordss [,loc]]]
A pointer to a newly-allocated number is placed in dst, containing a subvector of the source number referenced by src, of length nwords and starting at offset (which default to 1). The error jump is taken if an attempt is made to copy beyond the end of the source.

An object created by this operation will not necessarily have the same flags as its source. Only the object type flag (**F$NUM** or **F$VEC**) will be set.

As with string operations which copy, the source operand may be a transient pointer.

9.5.10 Conversion

CVTmn src, dst
Convert the source operand from mode m to mode n and place the result in the destination operand.

CVTSm src, dst[, loc]
Convert the source operand from a string to the specified mode. If this conversion is not possible, jump to loc.

CVTmS src, dst
Convert the source operand from its mode to a string. Place a pointer to this string into dst. Note that this implies a call to the storage allocator.

CVTID dst [,loc]
Convert an integer between 0 & 9 (inclusive) to an integer representing that digit in the local character set. In ASCII, this means add 60 (octal). If the error jump is specified, branch if the conversion cannot be performed.

CVTDI dst [,loc]
Convert an integer representing a digit in the local character set to an integer between 0 & 9. In ASCII, this means subtract 60 (octal). If the error jump is specified, branch if the conversion cannot be performed.

9.5.11 Symbol table manipulation

In the following ,note that the ´address of the entry´ is a pointer to a symbol table entry of implementation-defined format, and thus should only be manipulated by **GETVAL**, **SETVAL**, **GETFLG**, **SETFLG**.

Also, the name table operands (nt) are pointers to hash tables of the format described in Section 9.4.

Alphabetic case is not significant in the operations which search name tables.

FIND nt, src, dst
The source operand is a pointer to a string. The name table specified by nt is searched for that name; if the name is found, the address of the entry is placed into dst; otherwise, dst is set to null.

ENTER nt, src, dst [,loc]
The source operand is a pointer to a string, which is inserted into the name table specified by nt. The address of the entry is placed into dst. If loc is specified and the returned entry existed previously (i.e. the symbol was already in the table),jump to loc. This jump is not a true LIL error jump as the error registers are not affected. If loc is missing, always continue with the next instruction.

Note that a successful **ENTER** will call the storage allocator. The newly-created table entry will have a value field of NULL and a flag field of zero.

REMOVE nt, src [, loc]
The source operand is a pointer to a string. The name table specified by nt is searched for that name; if the name is found, the entry and its name and value pointers are deallocated. If the name is not found, take the error jump if specified.

GETVAL src,dst
The source is an 'address of entry' returned by FIND or ENTER. The value pointer of the source is placed in dst.

GETFLG src,dst
The source is an 'address of entry' returned by FIND or ENTER. The flag word of the source is placed in dst.

GETNAM src,dst
The source is an 'address of entry' returned by FIND or ENTER. The name pointer of the source is placed in dst. Note that the table entry still retains 'ownership' of the referenced object.

GETNXT dst [,loc]
The destination is an 'address of entry' returned by FIND or ENTER, which receives a pointer to the next entry with the same hash code. Take the error jump if the destination originally contains the null pointer.

SETVAL src,dst
The destination is an 'address of entry' returned by FIND or ENTER. The value pointer contained in the source is placed in the symbol table block pointed to by dst. Note that this instruction preserves the 'one-owner' discipline,i.e., the symbol table block becomes the 'owner' of the value and the former value is deallocated.

SETFLG src,dst
The destination is an 'address of entry' returned by FIND or ENTER. The flag word contained in the source is placed in the symbol table block pointed to by dst.

9.5.12 Heap object prefix manipulation

GETTYP src, flag
The flag field of src is placed into flag.

GETLEN src, dst
The length (in bytes or words, depending on the type) of src is placed in dst.

SETTYP flag, dst
The flag field of dst is replaced by flag. Note that this is an extremely dangerous operation, and some implementors may want to restrict some bits from manipulation.

SETLEN src, dst
This instruction is not recommended for implementation.

9.5.13 Stack operations

PUSHm src
Push src onto stack. Note a typeless form exists for register operands.

POPm dst
Pop top of stack into dst. Note a typeless form exists for register operands.

PUSHM Rm,Rn
Push register Rm through Rn onto stack. Note that this operation is typeless.

POPM Rm,Rn
Pop stack into registers Rm through Rn.

9.5.14 Routine calling

CALL locl [,loc2]
Save the next instruction address on the stack, and transfer control to locl. If loc2 is specified,jump there on a failure return (initiated by the FAIL operation).

PCALL locl,loc2
Similar to the CALL instruction, but intended for calling LIL subroutines which perform the action of LOGO-S primitives. Implementations with limited memory or address space may use this instruction to invoke an overlay handler or a context switch. The failure jump address is mandatory, as all LOGO-S primitives may fail as the result of an interrupt. The interrupt status word is polled both before and after calling the primitive. The failure jump will be taken if an interrupt is pending.

RETURN
Pop the top element of the stack, and place it in the program counter. This operation is meant to return from a transfer initiated by the CALL operation with one operand.

FAIL
Pop the top element of the stack, take failure return previously specified by the two operand form of CALL or PCALL.

SUCCEED
Return from a transfer initiated by a two operand CALL or PCALL.

HOST addr
Call a host environment (non-LIL) routine specified by addr, with the LIL registers used to pass arguments and receive results. Such a call may not necessarily affect the (system) stack. Note that addr is not necessarily a machine address and can only be a single symbol, not a general address expression.

The defined host routines are described in Chapter 10.

9.5.15 Flag operations

A flag is defined as an integer of which only the rightmost 16 bits are significant.

TSTB flag,dst,loc
Jump to loc if all the bits which are on in flag are also on in dst.

SETB flag,dst
Turn on all bits in dst which are on in flag.

CLRB flag,dst
Turn off all bits in dst which are on in flag.

9.5.16 Error Instructions

SETERR code,loc
Place code and the program counter in the error registers, and transfer control to loc. This action of this instruction is equivalent to the action which occurs upon the recognition of an error condition in a LIL instruction.

GETERR code,dst
Move the contents of the error registers into dst and code.

9.6 The LIL assembly language

9.6.1 Basic structure

The LIL assembly language has been designed to be mapped very easily onto a given target assembly language. However, it is expected that some amount of preprocessing will be necessary in order to render a LIL source program acceptable to most assemblers.

A line of LIL text consists of four fields, as follows:

```
    [label:] [opcode] [operands] [;comments]
```

any number of spaces or tabs may separate the fields, but the label, if present, must start in the first character position of the line. If no label is present, the first character position must be blank.

LIL symbols may be of any length, but only the first six characters are considered significant. Thus, if your assembler restricts symbols to, e.g., eight characters, the preprocessor could throw away all but the first eight characters of the symbol, with no loss of meaning. A symbol begins with an alphabetic character or a dollar sign, followed by a sequence of alphabetics, dots, dollar signs, or digits. Case is not significant--lower case letters are treated in identical manner as upper case letters.

9.6.2 Assembly expressions

A simple expression consists of one of the following elements:

1. a symbol: the value of the symbol is used.
2. an integer: the integer´s value, base-10, is used.
3. a character, enclosed in double quotes: the character´s code value is used. Note that it is a heinous crime to attach significance to the numeric value of the character code.

Compound expressions are formed by connecting the above elements by the operators +, -, and *, with the usual meanings. Parentheses and other operators are not available. Compound expressions may appear only in assembly directives, not in LIL instruction operands.

As assemblers vary widely in the types of relocatable expressions permitted, no advice is given here. To be safe, expressions should be either absolute or simply relocatable. Expressions used in data reservation directives must be absolute.

9.6.3 Address Expressions

The following is a description of the standard LIL address expressions, i.e., the forms which an instruction operand may take:

- Register mode operands are one of the symbols R0, R1, R2, R3, R4.
- Memory mode operands are any other single symbol, e.g., FOO.
- Immediate mode operands are of the form
  <#,EXPR>
  where EXPR is any simple expression, e.g., <#,3>.
- Indirect mode operands are of the form
  <OFFSET,REG>
  where REG is an integer between 0 and 4 specifying a register,

and OFFSET is either a simple expression or the symbol @ (which means an offset of zero). An example is <TABLE,3>.

9.7 LIL Assembly Directives (Pseudo-Operations)

9.7.1 Preprocessing Directives

CONST name,value
Defines the name to be a manifest constant with the given value, which may be any expression.

INCLUDE filename
Inserts the contents of the specified file in place of the INCLUDE statement. The included file may only contain symbol definitions (may not generate any code). On some systems, it may be necessary to assemble the file to be included previously.

9.7.2 Program Sectioning Directives

ENTRY name, name, ..., name
Defines the specified names (defined elsewhere in this module) as global.

EXTRN name, name, ..., name
Defines the specified name (used within this module, but not defined) as global.

MODULE name [,INCLUDE]
The first statement (apart from listing control) of any module, this assigns a module name, and does any peculiar things needed by the loader in question. The use of the optional keyword INCLUDE declares this module to contain only symbol definitions and may be used as the argument of the INCLUDE directive (in another file).

ENDMOD name
This is the last statement of a module. The module name is given for clarity's sake.

PSECT
This instruction specifies resumption of the program section; succeeding statements are assembled immediately after the last program section code. The program section may be read-only.

VSECT
This instruction specifies the code following to be assembled into a variable (or write-enabled) section; succeeding statements are assembled immediately after the last variable section code.

9.7.3 Data Generation Directives

BYTE exp,exp,...,exp
This instruction assembles a sequence of integers into successive bytes.

TEXT "characters"
Starting at the next word boundary, this instruction assembles the characters into successive bytes. Any trailing bytes before the next word boundary are filled with zeroes.

PFIX type,length
This instruction creates a "pseudo heap object". A prefix is assembled for a heap object of type **type** with a logical length of **length**. The type field will contain the bits specified by **type**. The resulting data structure is assembled in-line. Subsequent data generation or reservation pseudo-ops are assembled into the "pseudo-heap object" until the logical length (specified in the **PFIX** directive) is reached. A **PFIX**, **TEXT** sequence can be useful for creating messages.

PTR exp, exp, ..., exp
This instruction assembles a sequence of pointers with the specified values.

INT exp, exp, ..., exp
This instruction assembles a sequence of integers with specified values.

BLKP exp
This directive reserves a block of **exp** pointer cells.

BLKI exp
This directive reserves **exp** integer cells.

PRIM procname,flag[,procaddr [, syment]]
This is equivalent to the following sequence of LIL pseudo-ops:

```
            extrn   procaddr, procname, syment
            ptr     procname
            int     flag
            ptr     procaddr
            ptr     syment
procaddr and syment default to 0 if not specified.
```

The first three operands are the components of a symbol table entry. **syment** is the address of a cell in which to store the address of the entry.

9.8 Coding conventions

We have evolved a style of LIL coding which follows a certain standard. Specific issues include register preservation and restoration, routine structure, and error detection.

9.8.1 Register usage

Generally speaking, R0 is used as the result register from a routine, with arguments being passed in R1, R2, ... Routines are generally permitted to modify input registers (however, host routines may not).

Responsibility for saving and restoring registers lies with the called routine. Such routines should preserve any registers they use.

An implicit distinction is made between "local" and "global" routines. If a routine is only called by one other routine, in the same module, then any register conventions are purely local. Thus, for example, no register saving or restoring might be done. Global routines, however, must leave registers preserved.

There is also a distinction made between "unsafe" and "safe" routines, i.e., those which do or do not call the storage allocator. Because R4 is in use most of the time as the frame pointer and thus may, in many implementations, define the extent of available free storage, "unsafe" routines should always make sure that R4 has the appropriate value whenever the storage allocator is invoked. "Safe" routines may use R4 at will as long as it is restored upon routine exit.

The system stack is not assumed to be available to the storage allocator. Thus pointers to objects within the heap should not be pushed onto the stack inside "unsafe" routines.

9.8.2 Conventions for LOGO-S Primitive Routines

Routines which carry out the action of a defined LOGO-S primitive have a special convention for passing arguments. R1 is assumed to contain the number of arguments passed to the primitive. R2 points to a buffer containing the arguments. The arguments are so that the first is accessed indirectly through R2 and the following arguments are stored in the consecutive words below the first. Thus, succeeding arguments may be obtained by decrementing R2. Normally, this buffer is a part of the LOGO stack (see Section 11.6). In some routines, in which the number of arguments is fixed, R1 is ignored and used as a temporary register.

Arguments in the buffer need not be preserved unless an error is detected. In that case, the LOGO error handler needs the original contents of the buffer. R1 and R2 need not be preserved as they are assumed to be saved by the calling routine.

9.8.3 Error handling

One significant feature of LIL is the concept of a failure return from a procedure. Almost universally in the portable LOGO system, a procedure which fails is taken to have detected an error. Such failures are propagated back to the evaluator, which then signals the error. Thus, there are large numbers of code sequences such as:

```
        call    mysub, failoc
        .
        .
        .
failoc: fail
```

The exception to this policy is in the parser. Because recursive descent is used, there are routines which simply fail to detect the non-terminal being searched for. Their callers use failure to signal that another alternative should be attempted. (If a genuine parse error is found, a variable in the parser is set appropriately; the top-level routine examines this variable to determine the failure type).

10. THE HOST ENVIRONMENT

The initial task of the implementor is to provide an interface program, written in whatever language the implementor wishes (although it will generally be either assembly language or some low-level system programming language such as Bliss, BCPL, or C), which has several responsibilities. It must provide a number of defined routines; these include:

- terminal input/output.
- file input/output.
- timer, attention, and program interrupt handling.
- mathematical functions.

In addition, it must receive control from the operating system, initiate and terminate an invocation of the LIL machine and provide debug facilities for the implementor. Finally, it must provide dynamic storage allocation for the LIL machine.

10.1 The Storage Allocator

The storage allocator is the most critical part of the host environment. The implementor must be sure that it functions correctly before integrating it with the rest of system as allocation bugs are extremely difficult to detect. Fortunately, the one-owner discipline minimizes the requirements for the allocator.

The design of such a storage allocator is quite simple, since the need for "garbage collection" (the identification of objects to be returned to free store) becomes unnecessary. The implementor must understand the "one-owner discipline" well in order to write the allocator. Although storage management in detail is handled by the implementation, LOGO-S has certain expectations regarding the methods by which storage is allocated and freed. The mechanisms by which the heap is maintained may vary from one implementation to another, but in broad outline things will stay quite constant.

The first point to note is that storage is indeed allocated dynamically; the exact demands made upon the heap will vary not only from program to program, but also from time to time when running the same program. This immediately rules out the sort of static, compile-time, allocation generally done by Basic or Pascal systems.

The traditional methods of storage allocation can be divided into two classes: programmer-specified and automatic. In the former class, one generally finds environments in which explicit allocate and free primitives must be performed. The latter class includes environments in which an allocate primitive is used, but "dead" objects are reclaimed by the system, by means of reference counts or garbage collections.

Both of these strategies have very serious problems: requiring the programmer to specify in detail each heap transaction means that he/she will inevitably commit one of two varieties of sins, either in freeing a precious piece of data, or in forgetting to free junk. Automatic strategies minimize such errors, but have other problems, since a reference count scheme cannot detect junk circular lists, and automatic garbage collection causes the system to go into a coma from time to time.

To be sure, there have been clever fixes proposed: for example, incremental garbage collection schemes have been suggested. However these are extremely complicated (thus making bugs very difficult to find), and impair the running efficiency of the program.

The solution adopted in portable LOGO is a combination of the two strategies. If each object has exactly one reference to it, then a reference count scheme can be implemented trivially: when a pointer is changed to refer to a new object, the one it referenced before is now obsolete, and may thus be immediately reclaimed. Such a scheme prohibits circular lists, and, furthermore, requires no reference count which could overflow. As we said earlier, the mechanics by which this is done are not precisely specified; implementations with virtual memory or limited memory may employ differing tactics.

We can, however, present a few recommendations. First of all, we suggest that the prefixes of heapjects contain a backpointer field, i.e., a pointer back to the cell which owns the object. This means that allocated objects can be moved within the heap quite easily as the contents of the ´owner´ may be updated immediately. Thus, ´compaction´ is trivial. This probably means that complicated schemes for consolidating free storage and determining where in the heap to allocate new objects should be avoided, since the overhead of ´unecessary´ compactions will be low. (However, systems with a paged memory may want to spend a little extra effort to reduce the working set size). The implementor should be aware that the net effect of the one-owner discipline is to spread the overhead of maintaining the heap over the entire execution of a program, and the design of the storage allocator should reflect this.

The placement of the heap in memory is not specified. We recommend the classic arrangement whereby the heap grows towards the LOGO stack (described in Section 11.6). The placement of the pre-allocated space for name table entries should be adjacent to either the heap or the stack, either one of which can be shifted to allow for name table growth.

(The name table entries can never be shifted as they do not follow the one-owner discipline).

10.2 Host interface routines

This section describes the routines needed to interface LOGO-S to a given host environment.

The routines all have names prefixed by a dollar sign. This method has been chosen to make host calls stand out. If a given assembler does not accept the dollar sign in symbols, the $'s may be edited out (by the preprocessor) with no loss of information. In general, any symbol beginning with a dollar sign is known to the host interface.

The routines are described here in a relatively high level notation, $foo(bar, zot, goo). By convention, the parameters to a host routine are placed in R1, R2, ..., and the value is returned in R0 (some routines return a value in R1 as well).

10.2.1 Input/output

The input/output system for portable LOGO is a compromise between more or less contradictory goals of easy implementation on the one hand, and efficiency and access to host environment facilities on the other. The resulting system is character oriented--the efficiency of a host system call renders procedure call overhead negligible.

Files are represented by an object termed herein an fd, or file descriptor. This object is a positive integer, and has an implementation defined meaning. The only requirement is that an fd uniquely identify a given opening of a file. Within these limits, an fd has whatever contents the implementor considers desirable. It may be a channel number, control block address, or even a Fortran logical unit number.

The end-of-line character (EOL), is used to represent whatever sequence of characters denote an end-of-line or end-of-record for a particular file, for both input and output. Thus, the writing of the EOL character to the fd representing the console, for example, will generally result in the actual transmission of a carriage return, line feed sequence, the translation being done by the host routine $wrch (see below).

10.2.1.1 $open(filename,mode)

filename is a pointer to a string which somehow identifies a file. Its exact format is implementation defined. **mode** is an integer which indicates whether the file is to be opened for input, output, append, random access, or interactive. The integer fd is returned. If the file cannot be opened the fd returned is negative, and its magnitude is an error code.

10.2.1.2 $close(fd, mode)

After execution of a call to this procedure, fd no longer refers to an active file. It is, at that point, an error to use fd as a parameter to an I/O routine--the effects are undefined. It is up to the implementor to determine whether closed fd values may be reused. Mode is an integer which determines whether the file is to be closed preserving changes made while open; closed and restored to its state before opening (this may not be possible on all systems); or, closed and deleted from permanent storage.

The system closes all active files when LOGO-S is terminated.

10.2.1.3 $rename(filename1,filename2)

The file previously identifiable by filename1 will henceforth be identified by filename2. (Both arguments are string pointers).

10.2.1.4 $wrch(fd, ch)

Writes the character ch to the specified file.

10.2.1.5 $rdch(fd)

Accepts one character from the specified file, and returns it. For files opened in interactive mode, $rdch is defined so that all previous characters passed to $wrch are actually written before $rdch returns. (This may necessitate writing out partially full buffers).

The occurrence of a keyboard interrupt causes the immediate return of $rdch with the value 0 (zero).

10.2.1.6 $sense(fd, item)

Item is a integer which is implementation dependent, but very likely to refer to a bit position in a file status word, or a word in a file descriptor block. The integer value of this item is returned. Currently, the following items are defined:

IO$PTR	Returns the integer offset of the current byte from the start of the file.
IO$EOF	Returns a 1 if the end of file has been reached, otherwise 0. This is not defined for interactive files.
IO$SIZ	Returns the total number of bytes in a file.
IO$ERR	Returns 0 if the file is not in error, otherwise some

	system-dependent number.
IO$WID	The column width of a line is returned. 0 returned means the line has infinite width.
IO$COL	This returns the current column position in the line. Zero is defined to be the left-most position.
IO$CHR	1 is returned if a character is immediately available, otherwise 0 is returned.

10.2.1.7 $control(fd, item, value)

The item field of fd is altered according to value. Currently, the items that may be set are IO$PTR (random access files only), IO$SIZ (for systems which need to pre-allocate space), IO$ERR (with a value of 0 to clear an error), and IO$WID (interactive files only).

10.2.2 Storage allocation

These routines may also be called implicitly by LIL instructions.

10.2.2.1 $ALLOC (size,type,pointer)

The storage allocator sets the contents of pointer to reference a heap object of the size and type given.

10.2.2.2 $DEALC (pointer)

The heap object referenced by pointer is returned to free storage.

10.2.2.3 $CLEAN

Allows the storage allocator to perform various system dependent functions. The main purpose of this call is to allow time-consuming functions of the allocator to be performed at convenient times, e.g., doing a heap compaction while in a console I/O wait.

10.2.3 Attention and clock interrupts

Attention and clock interrupts are handled by means of polling. Since LIL is generally implemented by macro definitions, an asynchronous LIL interrupt could cause disaster, by occurring in the middle of a code sequence which corresponds to one LIL instruction.

Polling takes place at strategic places in the evaluator, as well as in some primitives (generally those dealing with input/output). The treatment of interrupts after they occur is dealt with in the section on the evaluator. This discussion is concerned with the duties of the interface.

In order to make polling fast, the global cell named $PI (for Pending Interrupt) contains the type of an asynchronous interrupt detected by the operating system. $PI contains a "small" (less than 256), non-negative integer. It is possible to test whether any interrupts have occurred by the instruction:

```
    jmpnzi $PI, intrpt
```

Thus polling is fast and may be done at any convenient place. The LIL instruction **PCALL** implicitly tests the value of $PI.

At the present time, there are three defined host interrupts: clock, user attention, and panic button. The first is used in handling timer traps; the second is used with "ordinary" attentions; the third is a stronger attention which is handled separately in order to allow some sort of safety valve for misbehaving programs.

As an example, the implementor might allow control-T, control-X, and control-C as user interrupts, and arrange that two control-C's without any other intervening keyboard entries within five seconds cause a panic attention. User programs could trap the control-T and control-X interrupts without preventing an emergency escape.

Several host routines are defined to permit synchronization between LOGO-S and the interrupt system of the host operating system:

10.2.3.1 $POLL()

Ensures the $PI word is up-to-date.

10.2.3.2 $RINT(inttyp)

$RINT is called whenever the LOGO machine begins to service an interrupt. All further interrupts of type inttyp are prevented until a $DINT (inttyp) is performed. A specific integer code associated with inttyp (for example, the ASCII value of the key which caused an attention interrupt) is returned in R0.

10.2.3.3 $DINT(inttyp)

The argument to $dint is an interrupt type, which informs the operating system that LOGO-S is through servicing that interrupt.

10.2.3.4 $CLINT()

Clears all pending interrupts.

10.2.4 Program check handling

The implementor should, if possible, trap program checks. Almost always a program check indicates a catastrophic error in the implementation. The system should print a polite message, and the call ´giveup´.

10.2.5 Timing

10.2.5.1 $elapsed()

Returns the amount of elapsed time from some relatively arbitrary point, as a pair of integers, the number of minutes and seconds, respectively.

10.2.5.2 $time()

Returns the time of day as a sequence of four integers which are the hour (24-hour clock), minute, second, and millisecond, respectively.

10.2.5.3 $date()

Returns the date as a sequence of three integers, the year, month, and day of the month, respectively.

10.2.5.4 $delay(n)

Although this is not properly a timing routine, this is the best place to describe it. n is an integer number of seconds; $delay returns when n seconds have elapsed.

10.2.5.5 $clock(n)

A clock interrupt is generated after n milliseconds.

10.2.6 Floating Point Arithmetic

Host routines are defined to convert LOGO number vectors into the local floating point format, and vice-versa, for calling local floating point higher-level mathematical functions.

These routines all signal errors by setting the $PI word for an evaluation interrupt. The routine $RINT returns the code of the error. Results are returned in registers 0 and 1, to allow for greater precision on some machines.

10.2.6.1 $CVTNF number

number is a number vector in LOGO floating point format (not yet defined). A floating point number is returned in R0 and R1.

10.2.6.2 $CVTFN float

float is a local floating pointer number in R0 and R1. A number vector is returned.

10.2.6.3 $SQRT float

The square root of the floating point number in R0 and R1 is returned in R0 and R1.

10.2.6.4 $SIN float

The sin of the floating point number in R0 and R1 is returned in R0 and R1. The input is in degrees.

10.2.6.5 $COS float

The cosin of the floating point number in R0 and R1 is returned in R0 and R1. The input is in degrees.

10.2.6.6 $TAN float

The tangent of the floating point number in R0 and R1 is returned in R0 and R1. The input is in degrees.

10.2.6.7 $ARCTN float

The arctangent of the floating point number in R0 and R1 is returned in R0 and R1. The output is in degrees.

10.2.6.8 $EXP float

e raised to the power of the floating point number in R0 and R1 is returned in R0 and R1.

10.2.6.9 $LN float

The natural logarithm of the floating point number in R0 and R1 is returned in R0 and R1.

10.3 Initialization and Normal Termination

The host interface begins execution of a LIL program by executing a sequence of three LIL-coded routines. These are responsible, in turn, for general initialization, "privileged" initialization, and running. The implementor may sequence the calls in any convenient way: thus, for example, the initialization routine may be overlaid.

General initialization sets up the heap appropriately, defines the primitives, and initializes the global communications area. The initialization routine is entered at the label $init.

"Privileged" initialization is only performed if the portable LOGO system is requested in a special (implementation-dependent) way. In general, privileged initialization will define special primitive functions which may be used for debugging, advanced system capabilities, or administrative purposes. It will also set an internal flag to indicate to the rest of the system that the user is privileged (whatever that may mean). The privileged initialization routine is entered, if appropriate, at the label $priv.

Finally, the main loop of the LOGO system is initiated by calling $start. Eventually, this routine may return; if so, the interface might return to the operating system, or might prompt for another login.

These routines are entered via normal LIL subroutine calls, that is, through the system stack. Thus, normal termination occurs when a corresponding return occurs, giving control back to the host system.

The implementor may choose to arrange matters so that some system (internal) failures result in a reload of the system. All of the portable LOGO routines ($init, $priv, and $start) must be serially reusable. If such a reload is performed, the global cell reload is set non-zero.

10.4 Storage Overflow

The storage allocator is defined to recognize situations in which store is dangerously low and set the $PI word accordingly. Normally, this will occur due to unterminated recursion in a LOGO-S program or an attempt to define an exceedingly large data object. Usually, the LOGO machine will be able to poll the $PI word and signal the LOGO error handler before all store is exhausted. (The LOGO-S user can trap this situation).

Occasionally, there will be a request to the storage allocator for more store than is available. If this occurs, the storage allocator should re-enter the portable LOGO system at the LIL-coded routine $bomb, after resetting the system stack and deallocating all heap objects not owned by symbol table entries or other heap objects (i.e., deallocate all objects owned by cells within the LIL code). $bomb will attempt to restore the top level of LOGO-S and continue.

10.5 Abnormal termination

Following the practice in TENEX and OS6, control paths that allegedly can't happen lead to calls to the routine giveup. This routine prints out a message, and then (optionally) enters the host system's debug routine. This is the only host routine which does not begin with a dollar sign.

11. GLOBAL DATA STRUCTURES OF THE LOGO MACHINE

In this chapter we describe the data structures available to all modules of the LOGO machine.

11.1 Words and Lists

The basic data types of LOGO-S - words and lists - are both represented in the LOGO machine as strings. Words are strings whose contents may be any sequence of bytes. Lists are distinguished from words by a flag (F$LIST) set in the string prefix. Lists use three special byte codes (the "untouchable" characters). U$SPCS separates the elements of a list, and U$LBRK and U$RBRK mark the beginnings and ends of sub-lists, respectively. The outermost brackets of a list are not explicitly stored.

This representation allows lists to be stored more compactly than would the usual network of pointers, because the one-owner discipline prohibits the use of common sub-lists.

11.2 Numbers

Numbers are internal data structures of the LOGO machine. They are created from strings by those LOGO-S primitives which output numerical results, and are converted back to strings by all other LOGO-S primitives.

Numbers are LIL numbers (vectors of integers). Each word, or digit, (except the first) contains a positive integer of lesser magnitude than the local base, with the lowest order digits stored in the words with the lower offsets. The first word of a number declares the number of words which describe the integer part of the number. (Currently, only integers are defined (fixed and floating point numbers will be included in a later version)).

11.3 Procedure Directories

A procedure directory is a vector containing the three significant components of a LOGO-S procedure: a representation of its title line (which include its local variables and formal parameters) its source code, and the parsed representation of the source code. The detailed structure of the last is deferred to the next section; for now, we shall consider the gross structure.

A procedure directory thus has the following format:

```
NINTS:  int 4+nlines
NLOCS:  int ?           ;number of local variables
NARGS:  int ?           ;number of arguments to procedure
NLINES: int ?           ;number of lines in procedure
LINENO: blki nlines     ;line numbers,stored as integers

ARGS:   blkp nargs      ;pointers to name table entries
LOCALS: blkp nlocs      ;pointers to name table entries

PARPTR: blkp nlines     ;pointers to parsed code vectors
                        ;if the NULL pointer, the line
                        ; has no current parse
SRCPTR: blkp nlines     ;pointers to source code strings
                        ;(these may be missing for
                        ; some procedures)
```

It is important to separate the source code from the object code; since the former may be thrown away, and since control sequencing concerns only the latter.

The prefix of a procedure directory contains flags which specify access to the procedure´s source code (read, write, parse).

11.4 Internal Program Representation

LOGO-S lines are parsed into an internal form prior to execution. This internal form is in the form of instructions to a hypothetical stack machine, the OM, or object machine. The action of this machine is simulated by the evaluator.

The OM format has been chosen in response to the following criteria:

- Since the original line is stored, there is no need to reconstruct source code. Therefore, the OM format is dictated entirely by considerations of ease and efficiency of interpretation.

- Although the surface level syntax of LOGO-S is fairly involved, the only real operations needed are for accessing of data, invocation of functions, and flow of control within a line.

- In order to simplify the definition of primitive functions, as much work needed for a function call is done by the evaluator and parser as possible. This means that tedious sequences for counting arguments and the like need not appear in each primitive.

The chosen representation bears a considerable resemblance to two popular postfix Polish architectures: the Burroughs B 5500 and the OCODE machine for BCPL. In this description OM instructions are described in a familiar assembly language form, that is, in the form:

```
    Instruction operand,operand,...
```

The OM instructions themselves are represented as pointers to the routines within the LOGO machine which simulate their defined actions. The operands are represented as pointers to the actual data objects.

11.4.1 Basic OM instructions

The simplest operation is that of pushing a literal onto the stack. This is accomplished by the "literal call" instruction litc. For example,

```
    litc "Foo
    litc [a b c]
```

pushes the two literals onto the stack. (Naturally, the literals themselves are represented by pointers. However, we will use their input form to represent them in this description.)

Accessing of variables could be accomplished by use of the literal call instruction followed by some sort of dereference operator. In fact, a dereference operator is supplied (see below). However, most variable accesses in LOGO-S are of the form ´:X´, which allow the parser to do the lookup of ´X´. Thus, a fair amount of space and time would be wasted by such a simplistic scheme. For this reason, the value call instruction, valc, is provided. For example, ´:X´ translates into

```
    valc <X>
```

(We use the form "<X>" to mean "the pointer to the name table entry for X".)

Forms such as ´::X´ are more complicated. ´:X´ outputs a literal, and therefore a lookup is needed. The dereference operator dref is provided to handle this situation. Thus, ´::X´ compiles into

```
    valc <x>
    dref
```

Procedures (and infix operators, see below) are called after their arguments are pushed onto the stack (as one might expect). The call instructions nominate the function and specify the number of arguments. (Argument counts are represented inside the parse vector as ´fake´ pointers, in this case, words in pointer form which are converted during the evaluation process into small integers.) This method is particularly convenient for greedy functions, which thus do not have to scan the stack. Considering the large number of greedy primitives, it is not surprising that this method was chosen. The instruction for this is ncal.

Note that no distinction is made between primitives and procedures, since the conventions for passing arguments are the same for both. Also, the distinction between ´normal´, ´greedy´ and ´quoting´ procedures has no significance for the OM machine. It expects that the parser will generate the correct code.

11.4.2 Control primitives

The most basic type of control primitive is that which ends execution of the current line. The eol instruction informs the evaluator that execution of the line is complete.

The eos instruction is generated at the end of each LOGO-S statement. It serves to signal the evaluator that the last procedure executed should have been a ´command´ (i.e. did not return a result). It initiates various clean-up operations on the stack.

The control statements require instructions for internal branching. Note that these instructions are not used for execution of the LOGO-S ´GOTO´ command, which requires that the named line be looked up in the procedure directory. Thus it is the case that the use of loop control statements is uniformly more efficient than the use of ´goto´, a situation which should bring joy to the heart of any structured programmer.

The simplest branching instruction is jump, which causes an unconditional branch to the instruction whose offset from the beginning of the line is given as the operand. Note that the use of offsets renders OM position independent.

The conditional branch instructions examine (and void) the top element of the stack, which must be either ´true´ or ´false´. (Any other value causes an error.) Because the evaluator needs to know (for error reporting) whether a conditional branch was generated by a LOGO-S **WHILE**, **IF**, or **UNTIL**, there are three such instructions: jmpi (IF), jmpw (WHILE), and jmpu (UNTIL). The first two instructions jump only if the stack contained ´false´, while jmpu jumps only if the stack contained ´true´.

The LOGO-S **REPEAT** statement requires a "decrement and jump" primitive. **REPEAT** first pushes the loop count onto the stack; the loop itself begins with

```
    decj Ln
```

where ´Ln´ is the address of the first instruction after the loop. (This address is represented as an offset within the parse vector). **decj** tests the top element of the stack. If it is zero, the jump is taken. Otherwise, it is decremented by 1, and the result pushed back onto the stack.

11.4.3 Infix operators

Rather than treat infix operators specially, the parser simply maintains a table of operators and associated procedures. Thus, for example, it is impossible to tell from the code

```
litc 3
litc 4
ncal <sum>,2
```

whether the user typed ´sum 3 4´ or ´3 + 4´. Since no unparsing need be done, there is no need for producing correct listings from OM code; further, the evaluator does not care which was typed. Thus, there are no OM instructions related to infix operators.

11.4.4 Examples

Here follow a few examples of LOGO-S lines and their translations.

```
longprint first :x butfirst [Michigan Huron Erie] 3
translates into:
valc <X>
ncal <first>, 1
litc [Michigan Huron Erie ]
ncal <butfirst>, 1
litc 3
ncal <longprint>, 3
eos
eol
```

The control instructions are given with labels of the form ´Ln´.

```
    if :x = :y ,
            print :x
    else {
            print :x;
            print :y
        }
    translates into:
    valc <x>
    valc <y>
    ncal <equivalent?>, 2
    jmpf L1
    valc <x>
    ncal <print>, 1
    eos
    jump L2
    L1: valc <x>
    ncal <print>, 1
    eos
    valc <y>
    ncal <print>, 1
    eos
    L2: eol
```

Finally, an example with loops.

```
while :x > 1,
        repeat 3, {
                   print "hello;
                   "x  <-  :x - 1
                  }
translates into
L4: valc <x>
litc 1
ncal <greater?>, 2
jmpf L1
litc 3
L3: decj L2
valc "hello
ncal <print>, 1
eos
litc "x
valc <x>
litc 1
ncal <difference>, 2
ncal <make>, 2
eos
jump L3
L2: jump L4
L1: eol
```

11.4.5 Summary of OM instructions

litc const	Push const onto stack.
valc name	Push value of name onto stack.
dref	Replace top element of stack by its value.
ncal name, num	Call function ´name´ with ´num´ arguments.
jump lab	Unconditional intraline branch.
jmpu lab	Branch if top stack element is ´true´.
jmpi lab	Branch if top stack element is ´false´.
jmpw lab	Branch if top stack element is ´false´.
decj lab	If top of stack is positive, subtract 1; otherwise branch to ´lab´.

eos	Signal command was executed.
eol	Terminate evaluation of the line.

11.5 Name tables

LOGO uses several LIL name tables in order to allow different classes of LOGO-S objects - procedures, variables, arrays, files - to share the same names, and to permit the fast lookup of literals used as arguments to primitives. The flag, name, and value components of a table entry have different meanings for each table.

In the variable name table, the flags indicate whether the name is being traced, or its use protected, and may be used in a future version for name control in parallel processing environments. The value field is a pointer to a LOGO-S string (the "thing" of the variable). An array name table entry is similar, except that the value field points to the vector which contains the array elements.

The procedure name table contains both the primitive and user-defined procedures. Flags indicate the trace status, and number of arguments for a procedure. They also indicate access to a procedure's source and object code. The value entry is a pointer to a procedure's directory, if it is user-defined, and to the address of its code, for a primitive.

A file table entry contains the file descriptor in the value field and uses the flags to indicate the mode of access. The flags for a special literal table entry specify the primitive functions for which the literal is an input, and the value is a small integer identifying the literal to the primitive.

11.6 The LOGO stack

The LOGO stack is maintained explicitly. Unlike the system stack whose top element only is accessible to the LIL programmer, the entire LOGO stack is generally available. The stack is organized into sections called frames. Each frame contains information specific to an active invocation of a procedure. The currently activated frame (the 'top' frame) is referenced by the frame pointer which is generally register 4 (R4). The stack grows 'down', i.e., each new frame is at a lower address in memory.

The contents of a frame are:

- The previous bindings of the arguments and local variables of this procedure. Note that, unlike a typical LISP implementation, the names of the variables are not kept on the stack. They are available through the procedure directory. The bindings are stored on the stack in the same order in which they occur in the directory.

- An integer containing flags describing the stack frame.
- A pointer to the symbol table entry of the procedure (this enables access to the procedure directory).
- The number of the line being currently executed within this procedure. This is represented as an integer offset to the line number stored in the procedure directory.
- An integer offset to the OM instruction currently being executed, within the vector representing the parsed line.
- A pointer to the previous stack frame. More specifically, this contains the previous value of the frame pointer.
- A ´local´ stack which is used for evaluating the OM code. This becomes the buffer for passing arguments for a primitive or procedure call.
- A pointer to the top of the local stack.

An important feature of the LOGO stack is that due to the ´one-owner discipline´ and the fact that a LOGO procedure (and its parsed OM code) may be invoked at any time, all literals contained in OM code must be copied onto the stack, and deallocated when their usefulness is ended. This is not necessary for variable bindings as their ownership is transferred between the stack and the name table.

12. THE LOGO MACHINE

The LOGO machine is the major piece of code distributed with the portable LOGO system. Its operation is described here mainly for the purposes of designing and debugging an implementation; it is not expected that the implementor understand it well enough to make modifications, except to the primitive functions.

12.1 Components of the LOGO machine

The LOGO machine consists of four major components: the parser, the evaluator, the primitive functions, and utility routines. A subsidiary component, the initialization routine, is used only at startup.

The parser accepts a string of text representing a program line. It outputs a sequence of instructions, represented as a LIL vector, for a hypothetical computer, corresponding to the intent of the program line. (Thus it should really be called a compiler.) It performs a recursive descent parse of the program line, emitting code as it goes--there is no parse tree.

The evaluator is the main control component of the interpreter. It obtains a line of LOGO-S code, represented as a string. It calls on the parser to process this code line, and obeys the resulting code string. If it is directed to perform a call to a LOGO-S procedure, it saves relevant state information on a stack, and then starts executing the lines of the procedure. If the procedure has been executed before, the code strings for procedure lines will have already been produced (and are stored with the original lines, which are retained for editing purposes). If the code string for a line has not been generated, the evaluator calls the parser, and stores the resulting code string (assuming no syntax errors) with the corresponding source line.

The evaluator is also responsible for handling LOGO-S errors; a comprehensive trap-handling system is provided to the LOGO-S programmer. When a trap is "sprung", by, for example, the failure of a primitive, the evaluator must determine whether a handler has been defined for the trap class, and, if so, pass control to the procedure defined as a handler.

The other task which falls to the evaluator is the invocation of primitive functions. When a primitive function is to be called, the evaluator produces a block of arguments, places the number of arguments and a pointer to the block in registers, and transfers control to a routine to handle the primitive function. Some primitives will share code; thus print and longprint will be handled by the same routine. Most of the primitive functions implement defined LOGO-S primitives; some additional primitives support the LOGO-S environment.

A number of utility routines are provided for the convenience of implementors. These are used to analyse and generate truth-values, to move LOGO data structures to and from the heap, perform arithmetic, and to do input/output.

12.2 The parser

12.2.1 General organization

The parser used in the Portable LOGO System is a top-down, recursive descent parser, and thus a straightforward implementation of the grammar found in the LOGO-S Reference Manual. It is the only module with recursive procedures.

12.2.2 Lexical scanning

The scanner sets two global cells: ´token´ receives a token type number, while ´tokval´ contains a string pointer to the actual token. For some tokens (e.g., parentheses, semicolons, and braces), no string pointer is needed. In such cases, ´tokval´ contains garbage.

Lists are read via the routine ´rdlist´, which is found in the reader. There is little point in duplicating that routine here.

All reading is done by calling the routine whose address is found in the global variable ´lexrch´. This routine is defined to read one character and return it. By doing this, use of the parser by ´run´ is facilitated.

12.2.3 Building object code

Object code is built into a buffer whose size is fixed. As the buffer is filled, it is written into the heap and emptied. Successive buffer loads are chained together using the buffer management utilities. Foward references are enabled using the routine ´mark´, which emits the special OM instruction noop to mark forward addresses, and the routine ´patch´ to replace noop with the address when it becomes available.

12.2.4 Infix operators

Infix operators are merely alternate representations of primitive functions. For this reason, the parser deals with an infix operator by merely looking up its associated primitive function and emitting a call to that function.

12.2.5 Error handling

The parser gives up on a line at the first syntax error. Since LOGO-S is intended to be used interactively, and since there are convenient line-editing facilities, we believe that the inconvenience of sometimes occasionally having to try a line three or four times is vastly outweighed by the terror induced in a naive user by a torrent of error messages which are totally meaningless.

The parser returns a list of tokens read and an error code. These are made available to the LOGO-S error trapper(see Section 13.2) to decipher the cause of the syntax error.

12.3 The evaluator

The evaluator is responsible for the sequencing of a LOGO-S program, maintaining the bindings of variables and the LOGO-S stack, and handling errors and traps. It invokes all primitives and user-defined procedures.

12.3.1 The operation of the evaluator

The evaluator ´executes´ a LOGO-S procedure in the following manner:

1. The evaluator obtains the address of the name table entry of the procedure to be evaluated from the global cell ´CURPRC´.

2. The evaluator obtains the arguments to the procedure. The local stack pointer of the previous frame is defined to point to the start of a block of argument bindings (on the stack). The evaluator obtains the addresses of the name table entries of the argument names from the procedure directory and sequentially exchanges the current bindings in the name table with the new bindings on the stack. (Thus, what was formerly the top of the previous frame now becomes the bottom of the new frame).

3. In a similar manner, the local variables have their name table bindings placed on the stack. Their new name table binding is the value ´UNDEF´ (for undefined).

4. The value of ´CURPRC´ and an offset in the procedure directory to its first line and the first instruction within that line are placed on the stack. These three values are referred to as the ´code pointer´.

5. The value of the frame pointer is placed on the stack (pointing to the previous frame). The frame pointer is now set to point

to this frame.

6. The local stack pointer is initialized on the stack.

7. If the current line is unparsed, the evaluator calls the parser with the source code of the line (this always happens for 'RUN procedures'). Newly parsed lines are stored in the procedure directory of the current procedure.

8. The evaluator simulates the action of the OM machine for each of the OM instructions generated by the parser, changing the code pointer as required. If an NCAL is encountered, the local stack pointer is set up to point to place on the stack where the result of the call is to be returned. If a user procedure is being called, the value of 'CURPRC' is changed and control transferred to the start of the evaluator. If a primitive is invoked, its arguments are deallocated by the evaluator upon return.

9. When the last line is evaluated (or a RETURN or DONE encountered), the result returned by this procedure is placed on the top of the local stack of the previous frame, the previous bindings of the local variables and arguments restored from the stack to the name table (and the current bindings de-allocated), and CURPRC and the frame pointer restored.

Certain primitives are actually part of the evaluator and are so marked in their procedure table entries. Instead of the normal subroutine linkage used for normal primitives, invocation of one of these primitives is merely an unconditional transfer to the appropriate location. Examples of these are: RETURN, DONE, GOTO, RUN.

12.3.2 Error handling

All errors, whether detected within the evaluator, the parser, or any other primitive set three global cells: 'ERRCOD', an integer labeling the error, 'ERRINF', a pointer to additional information about the error, and 'ERROBJ', a pointer to the offending object. Where appropriate, ERRINF contains a LOGO-S object which is a canonical representation of the LOGO-S expression which caused the error. For example, if the expression

```
    PRINT PRODUCT :X  SUM 7 1
```

were evaluated and the "thing" of X was "A, an error would be detected inside the primitive PRODUCT. The contents of ERROBJ would be "A (since it is non-numeric). ERRINF would contain the list:

```
    [ PRODUCT "A 8 ]
```

2

An error which occurs outside the evaluator is signaled by the failure of the detecting routine. Upon the failure return, the evaluator transfers control to the appropriate part of the error handler, which builds the information needed to process the error.

The error handler calls the trap handler. If a trap is enabled for the type of error detected, the evaluator begins evaluation of the trapping procedure. If not, the system gives up. Normally, the monitor always maintains traps for all errors. Upon return from the trapping procedure, the failed primitive is skipped, and the value returned by the trap is substituted.

Certain primitives poll the global cell $PI in order to detect asynchronous interrupts. If this occurs, the primitive fails in the same manner as if it had detected an error, although the information in the global error cells will, of course, be different. In particular, the cell ´ERRINF´ is supplied with the value of the top of the local stack, rather than with the normal context information, and ´ERRCOD´ is set to 0 to signal this. This informs the LOGO-S program that the failed primitive is to be retried upon a normal return from the trap, rather than skipped as usual. This also applies to OM instructions which fail, such as in the case of a non-numeric argument to decj or a non-boolean argument to jmpi.

12.3.3 RUN & VALUE

RUN is the LOGO-S primitive which takes a LOGO object and treats it as executable source code. (For example, all command lines typed into the LOGO-S monitor are executed by RUN). The evaluator treats the object which is the argument to RUN as a special type of procedure,i.e., a procedure with no arguments or local variables and only one line. A ´RUN procedure´ has a slightly modified format for its stack frame. The procedure symbol table entry is the value NULL, and the line number entry contains instead a pointer to the code to be executed. RUN is implemented as part of the evaluator.

VALUE is similar except that it expects to return a value, and thus can only evaluate expressions. EV$RUN and EV$VAL are the flags which determine if a stack frame with a NULL symbol table entry is an invocation of RUN or VALUE. VALUE signals an error if it evaluates a command.

12.4 The Trap Handler

All abnormal conditions detected by or reported to the evaluator cause the trap handler to be invoked. The trap handler's function is to check if there is a LOGO-S procedure prepared to handle the condition, and, if so, to set up a call to the trapping procedure.

12.4.1 The Trap Tables

The trap handler maintains two tables, TRPPRC and TRPSPC. TRPPRC contains pointers to a stack of trap data blocks, one entry per trap type (or channel). (A NULL entry means trapping is not enabled). Each data block contains the symbol table entry of the trapping procedure and the level of the stack frame which received the currently active interrupt (if zero, the trap is not currently active). The stacks of trap data blocks are maintained as a linked list of vectors with the following format:

```
    *****************
    *    prefix     *
    *****************
    *NINTS(always 2)*
    *****************
    * level of      *
    * stack frame   *
    * receiving     *
    * interrupt     *
    *****************
    * ptr. to symbol*
    *  table entry  *
    * of trap proc. *
    *****************
    *    link ptr.  *
    *****************
```

TRPSRC is a table of pointers to special routines for each type of interrupt. In the distributed portable LOGO system, the table is set up to merely return successfully from the trap handler. An implementor may change this table and add routines to the trap handler as desired.

12.4.2 Receiving an interrupt

Whenever the evaluator notices the $PI word is non-zero or an error occurs, it calls the trap handler with either the value of $PI or an error type as an argument. If the trap handler is unable to handle the trap specified, it fails and the interrupt is lost. (If this is bothersome, the implementor may, of course, write code to examine the trap tables before setting $PI). When successful, the trap handler sets up a call to the trapping procedure by placing its symbol table entry in the global cell 'CURPRC'. It then marks the stack frame of the current procedure, i.e., the one in which the trap occurred, with the trap type received. Control then returns to the evaluator.

The trap handler can handle a trap if it has a valid TRPPRC entry with a zero stack frame level, indicating that the trap is not currently in progress. Thus, interrupts of different types may be in progress simultaneously. However, an implementor may wish to enable some local priority scheme via the host calls $RINT & $DINT.

As noted earlier, some LOGO-S primitives (notably READCHAR) may detect the setting of '$PI' and fail (to avoid being caught in an I/O wait).

12.4.3 Dismissing an interrupt

A procedure called by the trap handler behaves exactly as if it had been through the normal LOGO-S control structure. Whenever the stack unwinds to the frame which is marked with a trap type, the trap is dismissed by the evaluator. A trap may also be dismissed by a call to the LOGO-S primitive CLEARTRAP. Dismissing a trap involves calling $DINT to inform the host operating of the dismissal, and setting the stack frame level entry in the trap data block to zero, and unmarking the stack frame.

12.4.4 TRAPRETURN

TRAPRETURN is a LOGO-S primitive similar to **VALUE**. It takes two arguments, however, a trap type and a LOGO-S object representing an unevaluated expression. If the trap type specified is currently in progress, the expression is evaluated and its value returned to the procedure (stack frame) in which the trap was sprung. This will involve unwinding the stack one or more frames, and dismissing the trap.

12.5 Initializing the LOGO Machine

The three routines '$start', '$init', and '$priv', located in the module 'LGOINI', initialize the LOGO machine and commence its execution.

12.5.1 Initializing the Name Tables

Primitive functions are defined by tables located in the module ´LGODAT´ and ´LGOINI´. These tables are rather large; the implementor may wish to arrange ´LGOINI´ so that this module vanishes, its space reused for something such as I/O buffers, or perhaps even more load table space, after use.

Each entry in the table is produced by a call to the pseudo-op prim (see Section 9.7.3). The routine ´loatab´ creates the name table entries from the load tables.

A few names (variables), files, and special literals used by LOGO-S primitives are also defined in this way. Each of these types also have their own table of ´prim´ pseudo-ops. In particular, the local system names of the file containing the bootstrapper and the console are in the file prim table; these will no doubt need modification.

12.5.2 Bootstrapping LOGO-S

$INIT opens the file ´BOOTS´, creates the first frame on the stack, and then passes a text string to RUN. This text string contains the LOGO-S code to read in contents of ´BOOTS´ and execute it. ´BOOTS´ contains code to read in the LOGO-S environment primitives from files and define them. The implementor will have to change the file name of ´BOOTS´ itself (in the prim table for files), and the names of the files opened by the code in ´BOOTS´ to suit the file naming conventions which he has chosen.

12.5.3 Creating a User Run Module

After returning from the bootstrapper, control is returned to the host environment. The state of the LOGO machine is such that a call to ´$START´ will allow normal operation of LOGO and prevent changes to the LOGO-S environment routines. Thus, at this point, the state of the machine may be saved in a run module, according to local system conventions. A call to ´$PRIV´ will allow the implementor to use the facilities of LOGO-S to modify the LOGO-S environment. These changes should be reflected in the bootstrapping files, and may be saved by running from ´$INIT´ again.

13. THE LOGO-S ENVIRONMENT

LOGO-S primitives are, for the most part, written in LIL. They may, however, be written in LOGO-S. Much of the programming environment, which is an integral part of a LOGO system, is indeed implemented as LOGO procedures.

There are several reasons for writing LOGO-S primitives in LOGO:

- A primitive may be naturally considered to be a composition of lower-level LOGO-S primitives, and thus more efficiently coded in LOGO-S. For example, **SAVE** and **GET** read and write special types of files, and thus can easily be composed of the basic file-handling primitives. In these cases, the primitive is either primarily used as a direct command (i.e., not in a procedure), or is interactive with the user, so that execution speed is not an important issue.

- Although we hope that most primitives will remain standard from system to system, there is likely to be some variation in the behavior of some of the surface-level primitives, especially in non-English implementations. Local modifications are most easily made in LOGO-S.

- On a system with limited amounts of primary memory, highly complex primitives may be more compactly stored as parsed LOGO code, rather than the equivalent LIL code. Also, some parts of the system which are not frequently used and interactive in nature may easily be overlaid as LOGO-S code, such as parts of the error reporting system.

Most of the LOGO-S source code comprising the LOGO-S environment is fairly "self-documenting", and is designed to be readily understood by implementors. A local implementor is welcome to change it at will, providing the functionality outlined in the LOGO-S glossary is maintained. The following sections describe the major components of the LOGO-S environment with respect to areas of possible local modifications.

13.1 The monitor

The monitor is a LOGO-S procedure provided with the Portable LOGO system which performs the top level functions, i.e., it prompts the user for input lines and obeys them. Its evaluation is automatically commenced after initialization. Many LOGO-S system variables (names) are local to the monitor, including **PROMPT** and **TOPLEVEL**. **PROMPT** is passed as an argument. **TOPLEVEL** is the LOGO-S recursion level at which the monitor is running. Thus, the LOGO-S command

```
UNWIND :TOPLEVEL
```

will abort any running procedure.

The monitor also sets up the standard error and interrupt trapping procedures via calls to **SETTRAP**. Any local system default traps can be set here.

13.2 The standard trapping procedures

13.2.1 The Error Reporting Subsystem

The error reporting subsystem uses the error trapping facility of LOGO-S to trap errors and produce messages. It is invoked by a call to **#ERRORTRAP**. The standard action for all errors detected by the LOGO-S environment is to print the appropriate system message and then unwind to **:TOPLEVEL**. However, if the user is currently in debug mode, **SUSPEND** is called instead of unwinding.

13.2.2 ERRORMESSAGE

System messages are returned by **ERRORMESSAGE**, called with the appropriate arguments (i.e., the arguments passed to the trap handlers). As suggested previously, much of the code for this may be maintained in secondary storage (perhaps stored in a random access file keyed by the error codes).

13.2.3 BREAK (SUSPEND)

The procedure **#BREAKTRAP**, invoked by a keyboard interrupt (usually set on the ASCII BREAK character), sets up a call to **SUSPEND**. **SUSPEND** calls a new monitor with a prompt extended by one ">".

13.2.4 PANIC

The procedure **#PANICTRAP**, invoked by a panic channel interrupt (usually set on the ASCII control-Z character), unwinds the stack to the current **TOPLEVEL**.

13.3 The Line Editing Subsystem (EDITSTRING)

The most obvious area where an implementor may wish to make modifications is in the input line editor **EDITSTRING** (which is also used by **REQUEST**). For example, on a system with all display terminals, one would expect the line editing facilities to be somewhat more sophisticated than those distributed with portable LOGO, which assume a full-duplex, hard copy terminal.

The LOGO-S line editing system comprises a great many routines. Each character the user types usually calls some other routine to carry out its function. Exceptions are the self-inserting characters and a few control characters (such as the end-the-edit character). Some naming conventions have been made for most of the routines called by **EDITSTRING**. Any routine beginning with either **EDIT#CAN#** or **EDIT#KILL#** cancels part of the line already typed, and preserves the cancelled text as explained below. Any routine that begins with **EDIT#NEXT#** keeps (types and skips over) part of the line. For instance, if one wished to modify the editor to do a backspace-space-backspace operation to delete characters on a video display, the routines **EDIT#KILL#CHAR**, **EDIT#CAN#CHAR**, **EDIT#CAN#WORD**, **EDIT#CAN#LOGLINE**, and **EDIT#CAN#BLINE** would have to be modified to suppress the backslashes they generate and to instead move the carriage.

Any routine name that does not begin with one of these type names usually does something special. These routines are **EDIT#RETYPE#LINE** (which retypes the current line, intended for hardcopy terminals but used also when a kill extends back across a line boundary to retype the current line) and **EDIT#SEARCH#CHAR** (which accepts a character and keeps everything after it). Notice that this routine resembles **EDIT#KILL#CHAR** in that while **EDIT#KILL#CHAR** deletes back to a certain character, **EDIT#SEARCH#CHAR** keeps up to that character. If **EDITSTRING** is modified for a video display, **EDIT#RETYPE#LINE** must still remain, but should instead clear the current line before retyping rather than give an initial newline.

13.4 The Procedure Editing Subsystem

The procedure editor is a collection of LOGO-S primitives which permit the user to examine and modify the set of procedures in the LOGO-S workspace. The procedure editor operates upon a privileged LOGO-S name (**EDITBUFFER**) which contains the current text of the procedure being created or modified.

The LOGO-S primitives **TO**, **TOGREEDY**, and **TOQUOTING**, are implemented as three LOGO EWOC procedures (for Eat Without Chewing), that is, procedures which accept an unlimited number of unquoted arguments. Each of these call **%#TO**. This routine puts the title line that followed into the editing buffer, and sets up the editing variables.

EDIT is also implemented as a LOGO-S procedure call. This procedure fills the editing buffer and sets up the LOGO-S names used by the editing procedures.

Any command line that begins with a digit is assumed by the parser to be a procedure line, and is converted into a call to **INSERT#LINE**, which inserts the text into the editing buffer in the proper numerical order. This routine detects an error if such a line is typed while not editing or defining any function.

END takes what is in the editing buffer and puts it back into internal storage by calling **DEFINE**.

#ALREADY#EDITING is a utility routine that informs the user that attempting to start another EDIT is impossible, as the user is already editing some procedure.

13.4.1 The Editing Variables

The variables used by the editor are all considered to be global; all but **#EDITBUFFER** are defined as local to the Monitor, and are initialized there. A brief summary of these global variables and their importance follows.

#EDITBUFFER is set to the current contents of the procedure being worked on, and changes with each edit to the procedure. Initially, it is filled with only the title line of a procedure that has just been specified with **TO**, **TOGREEDY**, or **TOQUOTING**, or, if the procedure was being edited, instead, it contains the entire text of the procedure (title line and normal lines). The contents of this variable are sometimes referred to as the editing buffer.

The procedure is stored as a series of lists. The first list contains the title line of the procedure. Subsequent lists hold two words each. In each list, the first word will be an integer specifying the line number, and the second word will be the text of the line.

#PROCNAME holds the name of the procedure either last or currently being worked on. This may be either from the user defining a new procedure or editing an old one.

EDITMODE is a non-privileged variable which tells the Editor that the user is either defining or changing a procedure. It is set to "True by **##TO** and EDIT, both of which put something into the EditBuffer.

##STARTDELETE is set to "False when the user has just inserted or kept some text, to indicate that the next kill clears the kill string. If "True, the kill string is not cleared; the user was just deleting before the current deletion.

##KILLED holds killed text, as a word in reverse order. The killed text gets tacked on to the end of the current line when the user restores killed text, and is not cleared until a delete after a non-delete (keep/insert).

##PRESERVED This is a string exactly like **##KILLED**, but its contents can only be changed by explicit action by the user. It is loaded from **##KILLED** in routine **EDITSTRING**.

##EDIT#CALLED is set in routine ETL (EDITTITLE), and cleared again in **##TO**. It tells **##TO** that **##TO** was called from ETL, and thus not to complain that the user is already in edit mode. This allows the user to completely change his title line.

##DELETING is set when a deletion is started, and cleared when a keep or insert is started. It is used to determine when to print opening and closing backslashes.

#REDEFINING is set to "False by **##TO** and to "True by **EDIT** to indicate whether a new procedure is being created or an old one is being modified. Currently, this variable is used only to tell the user, when he types **END**, that the procedure has either been defined or changed.

14. ADVICE TO IMPLEMENTORS

This chapter contains advice to a prospective implementor on strategies and tactics which minimize grief. Any contributions from implementors will be gratefully received and added to future additions.

One rather important point is that the text of the Portable System should not be capriciously modified. The intention of separating out the system facilities and making them into an interface is to preserve the portability of the remainder. Since updates and corrections will be distributed with respect to the original system, random local changes should not be made to that code.

One situation where such changes must be made concerns LIL program format. Many assemblers will have lexical requirements which differ from those of the Macro-10 assembler with which portable LOGO was developed. Many implementors will thus need to write preprocessors to perform format conversions such as the removal of colons after label names and even adjustment of fields to some local fixed field standard. It is crucial that the implementor treat the output of this preprocessor as transient (i.e., that the preprocessor be considered as a front end to the assembler). Patching preprocessor output is a recipe for disaster.

The PSECT and VSECT operations may be simply ignored. However, the implementor who wishes to have LOGO-S be reentrant (convenient on a time-sharing system) may choose to have PSECT and VSECT code assemble into different control sections , segments, or whatever the local linker calls them. This approach also has the advantage of allowing the implementor to arrange program and data sections in ways that minimize working sets (on virtual memory machines). The implementor is advised to study the capabilities of the available linker fairly carefully.

The "fake pointers" are best implemented as invalid addresses, if that is possible. In such a case, erroneous following of a fake pointer would cause an illegal memory reference.

APPENDIX A
LOGO BIBLIOGRAPHY

The following is a representative list of LOGO journal articles and other publications organized by institutional source.

BBN LOGO Group

Programming-Languages as a Conceptual Framework for Teaching Mathematics, W. Feurzeig and S Papert, Proceedings, NATO Science Conference on Computers and Learning, Nice, France, May 1968.

Programming-Languages as a Conceptual Framework for Teaching Mathematics, W. Feurzeig, et al., Final Report to National Science Foundation, NTIS No. PB-206919, June 1971.

Information Processing Models and Computer Aids for Human Performance, Task 3: Programming Languages as a Tool for Cognitive Research, W. Feurzeig and G. Lukas, BBN Report No. 2187, June 1971.

Computers and Learning to Learn, G. Lukas and W. Feurzeig, Proceedings of the Second City University of New York Conference, Innovations in Educational Technology, May 1972.

Uses of the LOGO Programming Language in Undergraduate Instruction, G. Lukas, BBN, Cambridge, Mass., Proceedings ACM National Conference, August 1972.

A Programmable Robot for Teaching, W. Feurzeig and G. Lukas, Proceedings of World Congress of Cybernetics and Systems, Oxford, England, September 1972.

Technology for Analysis of Student Interactions With Complex Programs, G. Lukas and W. Feurzeig, BBN Report BOLTBER-74-265, February 1973.

The Use of Dribble Files as Instructional Aids, W. Feurzeig and G. Lukas, LOGO Working Paper, August 1975.

Notes on LOGO Language Specifications for Portable Implementation, W. Feurzeig and G. Lukas, BBN, Cambridge, Mass., August 1977.

The LOGO Programming Language - Learning Mathematics Through Programming, W. Feurzeig, G. Lukas, and J. D. Lukas, ENTELEK, Newburyport, Mass. 1977 (Book).

Special Technology for Special Children, Computers to Serve Communication and Autonomy in the Education of Handicapped Children, E. P. Goldenberg, University Park Press, Baltimore, MD., 1979.

MIT LOGO Group

A Computer Laboratory for Elementary Schools, S. Papert, Computers and Automation, 21 (6), June 1972.

Teaching Children Thinking, S. Papert, Mathematics Teaching, No. 58, Spring 1972.

Twenty Things to do With a Computer, S. Papert and C. Solomon, Educational Technology, April 1972.

Teaching Children to be Mathematicians Versus Teaching About Mathematics, S. Papert, Int. J. Math. Educ. Sci. Technol., Vol. 3, 249-262, 1972.

On Making a Theorem for a Child, S. Papert, Proceedings of the ACM Annual Conference, August 1972.

NIM: A Game-Playing Program, S. Papert and C. Solomon, MIT LOGO Memo No. 5, January 1970.

Developing a Musical Ear: A New Experiment, J. Bamberger, MIT LOGO Memo No. 6, July 1972.

LOGO Manual, H. Abelson, N. Goodman, L. Rudolph, revised J. Adams. LOGO Memo No. 7, June 1974.

Use of Technology to Enhance Education, S. Papert, MIT LOGO Memo No. 8, June 1974.

TORTIS, Toddler's Own Recursive Turtle Interpreter System, R. Perlman, MIT LOGO Memo No. 9, March 1974.

Summary of MYCROFT: A System for Understanding Simple Picture Programs, I. P. Goldstein, MIT LOGO Memo No. 10, May 1974.

LLOGO: An Implementation of LOGO in LISP, I. Goldstein, H. Lieberman, H. Bochner, M. Miller, MIT LOGO Memo No. 11, June 1974.

LOGO Progress Report 1973-1975, H. Abelson, J. Bamberger, I. Goldstein, S. Papert, LOGO Memo No. 22, Revised March 1976.

Interim Report of the LOGO Project in the Brookline Public Schools: An Assessment and Documentation of a Children's Computer Laboratory, S. Papert, D. Watt, et al. LOGO Memo No. 49, June 1978.

Final Report of the Brookline LOGO Project, Parts I-III, S. Papert, D. Watt, A. DeSessa and S. Weir, September 1979.

Mindstorms: Children, Computers and Powerful Ideas, S. Papert, Basic Books (to be published in 1980).

University of California, Irvine
Department of Computer Science

Material and Ideas to Teach an Introductory Programming Course Using LOGO, G. Fischer, University of California, Irvine, June 1973.

Computers and a Liberal Education: Using LOGO and Computer Art, Ph.D. dissertation, R. Rubinstein, University of California, Irvine, 1974.

Recursive Functional Programming for the Student in the Humanities and Social Sciences, J. S. Brown and R. Rubinstein, Univ. of California, Irvine, Tech. Report #27a, Sept. 1974.

Using LOGO in Teaching, R. Rubinstein, ACM SIGCUE, Topics in Instructional Computing, Vol. 1, January 1975.

University of Edinburgh
Department of Artificial Intelligence

Using LOGO to Catalyse Communication in an Autistic Child, S. Weir and R. Emanuel, D.A.I. Research Report No. 15, January 1976.

Computational Metaphors for Children, J.A.M. Howe, T. O'Shea, D.A.I. Research Report No. 24, August 1976.

How to Work the LOGO Machine, Benedict du Boulay and T. O'Shea, D.A.I. Occasional Paper No. 4, November 1976.

Seeing the Works: A Strategy for Teaching Interactive Programming, Benedict du Boulay and T. O'Shea, D.A.I. Working Paper No. 28, May 1978.

Design of the TERAK LOGO System, P. M. Ross, D.A.I. Working Paper No. 49, February 1979.

Teaching Mathematics through LOGO Programming: An Evaluation Study, J.A.M. Howe, T. O'Shea and F. Plane, 1979.

Darmstadt
Forschungsgruppe CUU

Arbeitsmaterialien zum interaktiven Problemlosen mit Computerhilfe Band 2: Problemaufgaben, Teil 1: Mathematik, Teil 2: Linguistik, Teil 3: Informatik, Teil 4: Kunstliche Intelligenz, Heinz-Dieter Bocker and Gerhard Fischer, 1977.

Other Centers

Computers and Society, R. M. Baecker, University of Maryland, 1971.

An Informal Graphics System Based on the LOGO Language, W. M. Newman, Queen Mary College, London, England, National Computer Conference, 1973.

URI Call-OS LOGO Reference Manual, F. M. Carrano, R. E. Dyer, and S. W. Renuart, University of Rhode Island, February 1973.

A Machine Independent Implementation of LOGO, V. S. Manis, Thesis, University of British Columbia, October 1973.

Teaching and Learning Chinese in a Computer Environment, S. Poh, The Mitre Corp., October 1973.

Talking Music With a Machine, S. Beckwith, York University, York Gazette, Vol. V (1), September 1974.

Simulation and Animation, T.D.S. Duff, Thesis, Department of Computer Science, University of Toronto, January 1976.

Some Problems of Translating LOGO to a Romance Language, F. Curado and J. A. Valente, UNICAMP, Brazil, June 1978.

INDEX

, 138

! 138

" 137
"Invisible 120
"NoChange 120
"NoRead 120
"Primitive 119
"Stepped 120
"Traced 120

137
#Call 129
#Monitor 127
#Peek 129
#Poke 129

(138

) 138

* 60

+ 59

, 88, 89, 139

- 59

/ 60

: 137
:BackDeleteLogicalLine 133
:Delete 131
:DeleteRegion 131
:EndLogicalLine 137
:EOL 135
:Escape 139
:ForwardCharacter 135
:GetKilledText 133
:LastLine 128
:LF 135
:NextWord 136
:Null 130
:Precision 67
:PreserveKilledText 133
:RandomSeed 65

:RetypeLine 134
:SearchForward 136
:Suspend 130
:Tab 135
:TopLevel 92
:UntypePhysicalLine 132
:ViewPreservedText 134
:WordDelete 131
:YankPreservedText 134

; 139

< 42
<- 78
<= 42

= 38

> 40
>= 41

@ 137
:Space 134

Abbreviations:, definition of 28
ABC 54
AccessType 116
All 44, 45, 83
Altmode 139
Any 44, 45
Arctan 63
Array 83
Array, definition of 31
Array? 37
Arrays 83
ASCII 54
Ask 107

Back 121
BackDeleteLogicalLine 133
BackTrace 100
BF 50
Bk 121
BL 51
BN 47
Boolean values:, definition of 28
Both 44
ButFirst 50
ButLast 51
ButNode 47
Bye 127

C 46
Carriage return 135
Char 54

CharPointer 117
ClearScreen 125
ClearTimer 98
Close 115
Column 117
Command, definition of 30
Compact 49
Contexts, definition of 31, 81
Continue 103
Control B 133
Control D 131
Control delimiters, definition of 30
Control E 137
Control F 135
Control G 133
Control I 135
Control J 135
Control M 135
Control N 136
Control P 133
Control R 134
Control S 136
Control U 132
Control V 134
Control W 131
Control word, definition of 30
Control Y 134
Control Z 130
Coordinates 46
Coordinates? 36
Cos 63
Count 53

Date 128
Debug 101
Define 72
DefineArray 78
Delete 115, 131
DeleteRegion 131
Descriptors, definition of 31
Diff 59
Difference 59
Do 88, 89
Done 91
Dots 137

E? 35
Ed 73
Edit 73
EditLine 76
EditString 108
EditTitle 76
Edl 76
EdS 108
Edt 76

Either 44
Else 89
Empty? 35
End 77
EndFile? 116
EndLogicalLine 137
EOF? 116
EOL 135
Eq? 38
Equivalent? 38
Er 85
Erase 85
ErrorMessage 100
Escape 139
Escape character 139
Exp 64

F 50
False 43
Fd 121
Fetch 80
File 83
Files 83
FileSize 116
First 50
FirstPut 51
Fix 66
Fixed point numbers, definition of 58
Flat list, definition of 31
Flat? 36
FlatList? 36
Float 67
Floating point numbers, definition of 58
Floor 66
Follows? 43
Format 68
Forward 121
ForwardCharacter 135
Fput 51
FullPrint 110
FullWrite 114
FullWriteForm 55

GE? 41
Get 110
GetKilledText 133
Go 90
Goodbye 127
Greater? 40
GreaterEq? 41
Greedy procedures:, definition of 28

Heading 123
Hello 127
Here 123

Home 125
Horn 122

Id? 39
Identical? 39
If 88
Infix form, definition of 28
Input? 112
Int 66
Int? 36
Integer 66
Integer? 36
Integers, definition of 58
Intersection 49
Invisible 84, 120

L 50
L? 35
Lamp 122
Last 50
LastLine 127, 128
LastPut 51
LE? 42
Left 122
Less? 42
LessEq? 42
Level 92
LF 135
LineFeed 135
Lines 75
List 52
List, definition of 31
List? 35
Lists, definition of 46
LL 127
LList 52
Ln 63
LongList 52
LongPrint 109
LongProduct 60
LongSentence 52
LongSum 59
LongType 110
LongWord 52
Lower 55
LP 109
LProd 60
Lput 51
LS 52
LSum 59
Lt 122
LType 110
LW 52

M 53

Make 78
Member 53
Mod 61
Modulo 61

N 47
N? 36
Name 82
Name, definition of 31
Names 82
Neg 61
Negative 61
NextWord 136
NoChange 120
Node 47
NodePut 47
NoRead 120
Norm 65
Normal procedures, definition of 28
Normalization, definition of 57
Normalize 65
Not 44
NoText 73
NP 47
Null 130
Null character 130
Number, definition of 31
Number? 36

Object, definition of 31
Objects, definition of 46
Off 43
Old 76
On 43
Open 114
Operation, definition of 30

PenDown 122
PenUp 122
PopTrap 97
Precedes? 41
Precision 67
Predicate, definition of 28
Prefix form, definition of 28
PreserveKilledText 133
Primitive 84, 119
Primitive? 38
Primitives, definition of 28
Print 109
Procedure 82
Procedure? 38
Procedures 82
Procedures, definition of 28
Product 60
Property? 118

Protection 117

Quote 137
Quotient 60
Quoting procedures, definition of 29

R 90
Raise 54
Random 64
RandomDigit 64
RandomSeed 65
Read 111
ReadChar 112
ReadLine 112
Rem 60
Remainder 60
Rename 115
Repeat 90
Replace 76
ReplaceNode 48
Request 107
RequestForm 56
ResetTrap 98
Restore 76
Return 90
RetypeLine 134
Right 121
RN 48
Rq 107
Rt 121
RubOut 131
Run 92

S 52
Save 113
SaveFile? 37
SearchForward 136
Sentence 52
SetCharPointer 117
SetDifference 49
SetHeading 124
SetPrecision 67
SetProperty 118
SetRandomSeed 65
SetTimer 98
SetTrap 97
SetTurtle 123
SetX 124
SetXY 124
SetY 124
Sh 85
SharpSign 137
Show 85
Sin 62
Sort 55

SpringTrap 98
Sqrt 62
Statement, definition of 30
Step 102
Stepped 120
Store 79
SubObject 53
Sum 59
SuperQuote 137
Suspend 103, 130
Synonyms, definition of 29

Tab 135
Tan 63
Text 75
Text? 73
TextLine 75
Then 88, 89
Thing 78
Thing, definition of 31
Thing? 37
Time 128
Timer 100
Title 87
TitleLine 74
Titles 87
To 70
ToGreedy 71
TopLevel 92
ToQuoting 71
TouchBack? 126
TouchForward? 125
TouchLeft? 125
TouchRight? 125
Trace 101
Traced 120
TrapHandler 100
TrapReturn 99
TrapSet? 99
True 43
Ts 87
Type 110

Union 49
Until 90
Untrace 102
UntypePhysicalLine 132
Unwind 91
UnwindWith 91

Value 93
Version 128
ViewPreservedText 134

W 52

W? 35
Wait 128
While 89
Word 52
Word, definition of 31
Word? 35
WordDelete 131
Words, definition of 46
WorkSpace 84
Workspace, definition of 31
Write 113
WriteChar 113
WriteLine 114

XCoord 123

YankPreservedText 134
YCoord 123

Z? 37
Zero? 37

[137

\ 60

] 138

^ 138

{ 138

223

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Report No. 3968

THE LOGO-S LANGUAGE AND THE PORTABLE LOGO SYSTEM
VOLUME II: PROGRAM LISTINGS

Wallace Feurzeig
E. Paul Goldenberg
Robert Stachel
28 July 1980

The work described in this report was supported in part by the National Science Foundation under Contract No. SED-7611981. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Prepared by:

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

Prepared for:

The National Science Foundation

IR008520

SEP 2 1980

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

1.1 Volume 1 Errata, Corrigenda and Addenda 1
1.1.1 The Description of LOGO-S (Vol. 1, Part II) 1
1.1.2 The Guide to Implementation (Vol. 1, Part III) 1
1.2 Implementation Bugs and Deficiencies 2
1.2.1 Synonyms and Procedure Names 2
1.2.2 Known Bugs 2
1.2.3 Unimplemented Primitives And Features 3
1.3 Note to Implementors and Users 3

2. BRIEF DESCRIPTION OF THE DECSYSTEM-20 IMPLEMENTATION 5

2.1 Implementation Methods 5
2.2 Additional Information on the DECsystem-20 5

3. LISTINGS OF THE DECSYSTEM-20 LIL PROCESSOR 7

3.1 The LIL Macros 7
3.1.1 LILDEF.MAC 7
3.1.2 LILC.MAC 12
3.1.3 LILS.MAC 14
3.1.4 LILT.MAC 23
3.1.5 LILV.MAC 27
3.1.6 LIL.MAC 30
3.2 The Host Environment 47
3.2.1 LILA2T.MAC 47
3.2.2 LILALC.MAC 47
3.2.3 LILCMP.MAC 60
3.2.4 LILCNV.MAC 62
3.2.5 LILDAT.MAC 64
3.2.6 LILENV.MAC 68

    3.2.7 LILSYM.MAC 83
    3.2.8 LILVEC.MAC 88

4. LISTINGS OF THE LIL SOURCE FILES 91

  4.1 INCLUDE modules 91
    4.1.1 FLAGS.LIL 91
    4.1.2 CONST.LIL 93
    4.1.3 ERRORS.LIL 98
    4.1.4 TOKENS.LIL 101
  4.2 Executable Modules 103
    4.2.1 ARITH.LIL 103
    4.2.2 BUFMGR.LIL 121
    4.2.3 CKBOOL.LIL 130
    4.2.4 EVAL.LIL 131
    4.2.5 LGOARI.LIL 156
    4.2.6 LGOBOO.LIL 159
    4.2.7 LGODAT.LIL 170
    4.2.8 LGOGAR.LIL 184
    4.2.9 LGOINI.LIL 189
    4.2.10 LGOIO.LIL 198
    4.2.11 LGONAM.LIL 219
    4.2.12 LGOSTR.LIL 240
    4.2.13 LGOSYS.LIL 267
    4.2.14 PARSER.LIL 271
    4.2.15 READER.LIL 295
    4.2.16 TRPHDL.LIL 302

5. LISTINGS OF THE LOGO-S ENVIRONMENT 319

  5.1 Bootstrapping 319
    5.1.1 BOOTS.LGO 319
  5.2 The LOGO-S Library 321
    5.2.1 HACKS.LGO 321
    5.2.2 LIBRARY.LGO 321
    5.2.3 SPARE.LGO 345
  5.3 The Error Message Templates 354
    5.3.1 EVALUATOR.TRAP 354
    5.3.2 FILE.TRAP 355
    5.3.3 PARSE.TRAP 355
    5.3.4 TRACE.TRAP 355

1. INTRODUCTION

This volume contains the listings of the source code of the portable LOGO system now running at BBN on the DECsystem-20. In general, this system implements the LOGO-S version 0 described in the first volume of this report. Because BBN is continuing to develop LOGO, and because the portable LOGO system, like any other large and complex program, is continually being modified and debugged, the correspondence between these listings and the LOGO-S description is not exact. Therefore, we also include a list of all the known differences between Volume 1 and the current working version.

The listings are taken directly from the TOPS-20 files which are used to generate our system. Thus, they include some code which is specific to the DECsystem-20. Also, we include the MACRO-20 code used to implement LIL, for completeness, and to serve as a reference for other LIL implementations.

1.1 Volume 1 Errata, Corrigenda and Addenda

1.1.1 The Description of LOGO-S (Vol. 1, Part II)

- :NULL should be called :BREAK. (In the TOPS-20 implementation it contains the ASCII character NULL, but should contain whatever is the local break character).
- The predicates BOTH, EITHER, ANY and ALL actually are called BOTH?, EITHER?, ANY?, and ALL?.
- UNWIND 0 returns to the very top level (will get out of a suspend).
- The workspace primitive described as TITLES is actually invoked as ALL TITLES. (The use of the primitive or keyword TITLES is analogous to the use of PROCEDURES).
- There are two LOGO-S primitives not described in VOLUME 1. UNOPEN is similar to CLOSE except that it aborts the creation or modification of a file if possible under the local operating system. TEXTFORM is a transcription primitive similar to FULLWRITEFORM with the exception that sharp signs (#) are not translated to spaces, and at signs (@) are not inserted before punctuation characters.

1.1.2 The Guide to Implementation (Vol. 1, Part III)

There has been a significant change to the implementation of the LOGO-S environment (the LOGO-S library) from what is described in Volume 1. Although the broad outline remains the same, the LOGO library procedures called by LOGO-S primitives such as EDIT and EDITSTRING are changed greatly. In addition, there is a more general system for parsing the arguments to LOGO-S workspace primitives such as SHOW and ERASE.

Other changes are minor and documented within the listings.

1.2 Implementation Bugs and Deficiencies

1.2.1 Synonyms and Procedure Names

In general, synonyms are not yet implemented. Several primitives currently have different names or are called by one of their synonyms. They are (the current name is in parentheses): TEXTFORM (#TEXTFORM), FULLWRITEFORM (#PRINTFORM), REQUESTFORM (#PARSEFORM), ABC (ASCII), NEGATIVE (MINUS), EDITLINE (EDL), EDITTITLE (EDT).

Also, NUMBER? does not exist; however, INTEGER? is equivalent to NUMBER? in version 0.

1.2.2 Known Bugs

- Comparing words with lists in EQUIVALENT? and IDENTICAL? causes an error. Also, lists cannot be used in the other relational predicates (PRECEDES?, GREATER?, etc.)
- SUBOBJECT gives an "out of range" error for non-negative ranges which are outside the input object. It should return the empty object (i.e., consistent with ELEMENT).
- CHAR 0 gives an "out of range" error. It should return normally.
- POPTRAP only works for privileged users. Thus, users are unable to remove their own traps.
- ERRORMESSAGE prints its message, instead of returning it.
- LASTLINE doesn't return the last input line if an error occurred.
- A user may UNWIND, UNWINDWITH, or TRAPRETURN into an invocation of a privileged system procedure. This should cause an error instead.
- The evaluator does not invoke a panic stop if it is unable to trap an error. This means that errors within error handlers can cause disasters.
- Redefining a procedure to have a different number or type of arguments can cause disaster if there exist references to that procedure in parsed

(previously executed) code.

- SHOW sometimes FULLPRINTs when it should PRINT and vice versa.

1.2.3 Unimplemented Primitives And Features

- The following primitives and system names defined in version 0 do not yet exist: TRUE, ON, OFF, FALSE, RANDOMDIGIT, SETRANDOMSEED, :RANDOMSEED, INTEGER, GOODBYE, #PEEK, #POKE, #CALL, :BREAK, :ESCAPE.
- ERASE, SETPROPERTY, and PROPERTY? are not yet implemented but their supporting LIL-coded routines (ERASE#PROCEDURE, ERASE#NAME, etc.) do exist.
- There is currently no tracing or debug mode.
- There are no turtle primitives.
- >= and <= are not yet recognized by the parser.

1.3 Note to Implementors and Users

As we stated in Volume 1 of this report, we plan to continue development of LOGO-S and of the portable LOGO system. We will maintain future versions of the language and system documents and programs. thus, users and implementors are advised to write directly to BBN for current materials.

Inquiries and requests for the most current edition should be directed to:

BBN LOGO Project
Department of Educational Technology and Instructional Systems
Bolt Beranek and Newman, Inc.
50 Moulton St.
Cambridge, MA 02138

These continuing post-project developments go beyond the scope of the contract work and are not being supported by the National Science Foundation. They reflect the continuing interest of Bolt Beranek and Newman, Inc. in LOGO.

2. BRIEF DESCRIPTION OF THE DECSYSTEM-20 IMPLEMENTATION

2.1 Implementation Methods

The portable LOGO system now running at BBN is based on a straightforward implementation of LIL (described in Volume 1, Part III). The LIL language processor exists as a collection of macros in MACRO-20, the assembly language of the DEC-20. These macros are pre-processed by the MACRO-20 assembler into a machine-readable library. LIL source files can then be assembled by MACRO-20 by including a reference to the LIL library at the start of the file. (This appears as the non-LIL statement "search LIL" which the reader will notice in the listings).

Each LIL module resides in a separate TOPS-20 ASCII file. Those modules which are INCLUDEd in other LIL modules produce a TOPS-20 UNIVERSAL file as output. They must be assembled before the other LIL modules, which produce relocatable binary object code as output. UNIVERSAL files are produced by those modules which have the INCLUDE keyword in their MODULE statement.

The assembly of the LIL code contains external references, in the form of subroutine calls, to the host environment. This includes both the host routines (described in Volume 1) and other routines which support the more complicated LIL macros. The host environment is written in MACRO-20, with calls to the TOPS-20 monitor as needed.

The LIL object code and the host environment are loaded using the TOPS-20 utility program LINK, and the resulting core image can be saved and/or executed with the appropriate TOPS-20 commands. This core image is a version of LOGO without the LOGO-S environment, which is loaded upon execution. The LOGO-S environment exists as a collection of ASCII format disk files, representing LOGO command lines and procedures in a special format. When the environment is loaded, control will return to TOPS-20 and the core image may be saved as an executable LOGO-S interpreter (with the limitations mentioned in the previous chapter).

2.2 Additional Information on the DECsystem-20

For those readers not familiar with the DECsystem-20, the following manuals should provide the necessary background information for the program listings:

- DECsystem-20 User's Guide. Order Nos. AD-4179B-T2, AD-4179B-T1, AA-4179B-TM.
- Monitor Calls Reference Manual. Order No. AA-4166C-TM.
- DECsystem-10/DECsystem-20 Hardware Reference Manual Vol. 1. Order No. EK-10/20-HR-001.

- DECsystem-10 Macro Assembler Reference Manual. Order No. DEC-10-LMCOA-B-D.

They are available from:

Software Distribution Center,
Digital Equipment Corporation,
Maynard, MA 01754

3. LISTINGS OF THE DECSYSTEM-20 LIL PROCESSOR

3.1 The LIL Macros

This section lists the MACRO-20 files used to generate the definitions of the LIL instructions and assembly directives which are implemented on DEC-20. They all produce UNIVERSAL files as output from the MACRO-20 assembler. They are listed in the approximate order in which they must be assembled. The first file, LILDEF.MAC, is also used to supply constant definitions for the assembly of the host routines.

3.1.1 LILDEF.MAC

```
       universal LILDEF ;LIL global definitions  (not LIL macros)


        subttl opcodes for the PDP-10 version of LIL

        opdef .movei [movei]
        opdef .addi  [addi ]
        opdef .subi  [subi ]
        opdef .idivi [idivi]
        opdef .andi  [andi ]
        opdef .iori  [iori ]
        opdef .xori  [xori ]
        opdef .imuli [imuli]
        opdef .popj  [popj ]
        opdef .pushj [pushj]
        opdef .pop   [pop  ]
        opdef .push  [push ]
        subttl register assignments

;       temporary registers t-t2
;       in general,t & t2 may be used by any LIL macro or host routine
;       that wants them. t2 is defined to contain the remainder of an
;       integer division performed on t (which contains the quotient)
;       They are also the result registers with t being the preferred
;       register.
;       register 16 is used as a scratch register by LIL environment routines
;       written in RATMAC and thus should be left alone (NOT CURRENTLY)

        t=14
        t2=15
```

```
;       working registers a-e
;       a-e are mapped into 1-5 always and thus can be used for JSYS calls
;       They are preserved by all routines except $$ routines and where
;       explictly modified.
        a=1
        b=2
        c=3
        d=4
        e=5

        %err=13   ;temporary register for errors
        p=17      ;main stack pointer
        maxreg=12 ;highest register available to LIL macros
        lilreg=6  ;LIL register 0 is mapped into this value

        ;LIL register names

        lilr0=lilreg
        lilr1=lilreg+1
        lilr2=lilreg+2
        lilr3=lilreg+3
        lilr4=lilreg+4

        ;alternate names for registers
        r0=lilr0
        r1=lilr1
        r2=lilr2
        r3=lilr3
        r4=lilr4

        subttl parameters

        $psini=400000           ; start of program (pure) segment (psect)
        stksiz==1000;size of LIL stack
        ntabsz==2000;initial size of LOGO-S name table
        errsiz==10  ;size of error stack
        bytsiz==11  ;size of string bytes in bits
        wdsiz==^d36;no. of bits per word
        bytpw==wdsiz/bytsiz ;bytes per word
        rtbyte==wdsiz-<bytpw*bytsiz> ;bit position of rightmost bit in rightmost byte
        pfxsiz==2   ;size of data (heap) object prefix

        danger==1000            ;minimum distance allowed between stack & heap
                                ; (less than this causes compaction)
        warn==2000              ;minimum distance allowed before $PI is set
                                ; to allocator trap
;       IO modes
```

```
        iord=0
        iowr=1
        ioapp=2
        ioint=3
        iornd=4

;       Sense & Control items

        ioptr=1
        ioeof=2
        iosiz=3
        ioerr=4
        iowid=5
        iocol=6
        iochr=7

;       $close function codes

        iocls=0
        iodel=1
        ioabt=2
        subttl addressing mode symbols (values returned by makeaddr)

        immed==-1       ;immediate mode
        regis==0        ;register (direct) mode
        memor==1        ;memory (or register indexed) mode
        indir==2        ;register indirect or deferred mode

        subttl special symbols

        null=777777
        inflgs==0               ;initial flags for symbol table blocks

        strpnt==441100          ;left half of string pointer(used to create
                                ; string pointers from addresses)
        %free==400000           ;marks heapjects as free objects

;       special op codes for .JSERR jumps

        opdef   $JSERR [001000,,0]
        opdef   $LERR  [002000,,0]

        %jserr==1               ; codes for lookup in LILENV
        %lerr==2

        subttl data object type codes
        strtyp==1       ;string
        vectyp==2       ;vector
        listyp==4       ;list
        numtyp==8       ;number

        subttl error code generator

        define generr(errnam,errcod)
        <
        ifl <
         ifdef e.'errnam,<printx e.'errnam already defined>
         ifdef .e'errcod, < .e'errcod(errcod) >
            >

         e.'errnam=^D'errcod
         define .e'errcod(code)
         < printx code is defined for e.'errnam >
        >


        subttl global addresses

        extern  $start          ;starting address of LIL program
                                ; defined by 'start' pseudo-op
        extern  $errstk         ;LIL error stack
        extern  $stack          ;LIL stack

        extern  chntab,levtab   ;psi tables
        extern  $aprjp          ;address of APR service routine (user-defined)
        extern  intwrd          ;APR interrupt word--flags & pc

        extern  clktim          ; cell holding time of next clock int.
        extern  clknum          ; cell holding # of clock interrupts

        extern  rictab,softrp   ; interrupt ID routines and soft trap flag
        subttl USEFUL MACROS


        define getpnt(reg1,ptr)
;
;       creates string pointer to left-most byte of word pointed to by ptr
;        in reg1

<       ifdif <reg1><ptr>,< hrr reg1,ptr >
         hrli   reg1,strpnt
>

        define getlng(reg1,reg2)
```

```
;
;       moves logical length of object pointed to by reg2 into reg1
<
        hrrz    reg1,-2(reg2)
>

        define getlst(reg1,reg2)
;
;       moves pointer to 1st (byte,word) following object pointed to by reg2
;        into reg1
<
         move   reg1,-1(reg2)
>

        define getfgs(reg1,reg2)
;
;       moves flags of object pointed to by reg2 into reg1
<
        hlrz    reg1,-2(reg2)
>

        define .seterr(code,loc)
;
;       SETS error for LIL instructions
        <
         movei  %err,code
         movem  %err,$$ERCD##
         move   %err,.+2
         movem  %err,$$ERAD##
         jrst   loc
        >

        define .jserr(acc,loc)
;
;       handles a JSYS error. Uses error code found in acc
        <
         movem  acc,$$ERCD##
         movei  %err,.+3
         movem  %err,$$ERAD##
         jrst   loc
        >

        end
```

3.1.2 LIILC.MAC

```
       universal LIILC
;       LIL conversion macros

        define lil.c
<
;;      TRIVIAL CONVERSIONS
;;      trivial conversions are equivalent to MOV's if operands are different
;;       no-ops if operands are identical

        define trivial(src,dst)
         < ifdif <src><dst>,< .mov <src>,<dst> >
         >
;;

        SYN trivial,cvtic
        SYN trivial,cvtip
        SYN trivial,cvtci
        SYN trivial,cvtpi
        SYN trivial,cvtps
        SYN trivial,cvtsp

;; error codes --also used in LIILS

        generr(nns1,4001) ;;non-numeric string arg 1
        generr(nns2,4002) ;;non-numeric string arg 2
        generr(ndig,4003) ;;attempt to convert non-digit

;;      CVTIS--convert integer to string (allocate new data object)

        define cvtis(src,dst)
        <
         makeaddr(s1,src)
         makeaddr(d1,dst)

         ifl    swsl,<movei     b,opsl >
         ifge   swsl,<move      b,opsl >

         movei  c,opd1
         .pushj p,$$I2S##

        >;;end cvtis


;;      CVTSI--convert string to integer
```

237

```
;;              take error jump if non-numeric characters in string

        define cvtsi(src,dst,loc)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

        ifnb    <loc>,< makeaddr(ll,loc)
                        movei   t2,opll
                    >
        %%t==t
        ifl     swsl,<movei     t,opsl>
        ifg     swsl,<move      t,opsl>
        ife     swsl,<%%t==opsl>

        getlng  a,%%t
        getpnt  b,%%t
        .pushj  p,$$STI##
        $LERR   t2,e.nnsl           ;;error for non-numeric string
        movem   e,opdl
>;;end cvtsi

;;      CVTDI —convert (ASCII) digit to integer
;;              take error jump conversion not possible

        define cvtdi(dst,loc)
        <
         makeaddr(dl,dst)

         ifg    swdl,<%%t==t
                        move    t,opdl >
         ife    swdl,<%%t==opdl>

        ifnb    <loc>,< makeaddr(ll,loc)
                        movei   t2,opll
                        cail    %%t,"0"
                        caile   %%t,"9"
                        $LERR   t2,e.ndig
                        >
        .subi   %%t,"0"
        ifg     swdl,< movem    t,opdl >
        > ;;end of cvtdi

;;      CVTID — convert integer to digit

        define cvtid(dst,loc)
        <
         makeaddr(dl,dst)
         ifg    swdl,<%%t==t
                        move    t,opdl >
         ife    swdl,<%%t==opdl>

        ifnb    <loc>,< makeaddr(l1,loc)
                        movei   t2,opl1
                        skipl   %%t
                        caile   %%t,^d9
                        $IERR   t2,e.ndig
                        >
        .addi   %%t,"0"
        ifg     swdl,< movem    t,opdl >
        > ;;end of cvtid

>
        end ;lil.c
```

3.1.3 LILS.MAC

```
       universal LILS

        search LILDEF

        subttl LILS-- LIL string macros(KL version)

        define  lil.s
        ;contains string macros loaded into LIL.UNV via search LILS in LIL.mac
<
;;      LILS error codes

        generr(soe,1001) ;;String Offset Error

        ;; MOVS--copy string from src  to (allocated) dst.
;;                      (ignore loc)

        define movs(src,dst,loc)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)

        ifl     swsl,< movei    e,opsl >
        ifge    swsl,< move     e,opsl >

        getlng  a,e
        movem   e,MARK1##       ;;save adjustable copy of source ptr
        movei   b,strtyp
        movei   c,opdl
        .pushj  p,$$ALLOC##
        move    d,a             ;;length of string in a & d
        hrr     b,MARK1
        hrli    b,strpnt        ;;source string pointer
        getpnt  e,(c)           ;;destination string pointer
        extend  a,[ MOVSLJ
                          z ]
        halt    .               ;;halt if instruction failed
        setzm   MARK1
        >
        ;; CATS--copy concatenation of two strings to (allocated) dst

        define cats(src1,src2,dst,loc)
        <
         makeaddr(s1,src1)
         makeaddr(s2,src2)
         makeaddr(d1,dst)

        ifl     sws1,< movei    d,ops1 >
        ifge    sws1,< move     d,ops1 >

        ifl     sws2,< movei    e,ops2 >
        ifge    sws2,< move     e,ops2 >

        getlng  a,d             ;;source lengths
        getlng  t,e
        dmovem  d,MARK1##       ;;save sources in MARK1 & MARK2
        add     a,t             ;;length of destination
        movei   b,strtyp
        movei   c,opdl
        .pushj  p,$$ALLOC##
        hrr     b,MARK1
        hrli    b,strpnt        ;;string ptr. to 1st source
        getlng  a,b             ;;get 1st source length
        move    d,a             ;;a & d have 1st source length
        getpnt  e,(c)           ;;destination pointer
        extend  a,[ MOVSLJ
                          z ]
        halt    .               ;;halt if move failed
        hrr     b,MARK2##
        hrli    b,strpnt        ;;string ptr. to 2nd source
        getlng  a,b             ;;length of 2nd source
        move    d,a             ;; in a & d
        extend  a,[ MOVSLJ
                        z ]
        halt    .               ;;halt if move failed
        setzm   MARK1
        setzm   MARK2
        >;;end cats

;;      CMPS--compares two strings alphanumerically

;;      if src1 .gt. src2 jump to loc1
;;      if src2 .gt. src1 & loc2 is specified,jump to loc2
;;      otherwise,continue

        define cmps(src1,src2,loc1,loc2)
        <
        makeaddr(s1,src1)
        makeaddr(s2,src2)
        makeaddr(l1,loc1)

        ifl     sws1,< movei    a,ops1 >
        ifge    sws1,< move     a,ops1 >

        ifl     sws2,< movei    b,ops2 >
        ifge    sws2,< move     b,ops2 >

        .pushj  p,$CMPSTR##
        jumpg   t,opl1

        ifnb <loc2>,<   makeaddr(l2,loc2)
                        jumpl   t,opl2
                >
        >;;end cmps

;;      CMPSU--compares two strings alphanumerically without regard to case

;;      if src1 .gt. src2 jump to loc1
;;      if src2 .gt. src1 & loc2 is specified,jump to loc2
;;      otherwise,continue

        define cmpsu(src1,src2,loc1,loc2)
        <
        makeaddr(s1,src1)
        makeaddr(s2,src2)
        makeaddr(l1,loc1)


        ifl     sws1,< movei    a,ops1 >
        ifge    swsl,< move     a,opsl >

        ifl     sws2,< movei    b,ops2 >
        ifge    sws2,< move     b,ops2 >

        .pushj  p,$CMPOST##
        jumpg   t,opl1

        ifnb <loc2>,<   makeaddr(12,loc2)
                        jumpl   t,opl2
                  >
        >;;end cmpsu

;;      LOOK & VERIFY
comment ?
         define verify(src,brk,dst,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(bl,brk)
         makeaddr(dl,dst)


        ifl     swsl,< movei    a,opsl >
        ifge    swsl,< move     a,opsl >

        ifl     swbl,< movei    b,opbl >
        ifge    swbl,< move     b,opbl >

         move   c,opdl

         setz   d,

         .pushj p,$SEARCH##

         ifnb <loc>,< makeaddr(l1,loc)
                        jumpge  t,%label
                        .seterr e.otl,opl1
                  >
%label:  movem  t,opdl
        >;;end verify

         define look(src,brk,dst,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(bl,brk)
         makeaddr(dl,dst)
        ifl     swsl,< movei    a,opsl >
        ifge    swsl,< move     a,opsl >

        ifl     swbl,< movei    b,opbl >
        ifge    swbl,< move     b,opbl >

         move   c,opdl

         seto   d,

         .pushj p,$SEARCH##

         ifnb <loc>,< makeaddr(ll,loc)
                        jumpge  t,%label
                        .seterr e.otl, opll
                    >
%label:  movem  t,opdl
        >;;end look
?

;;      SUBSTR--copies nchar bytes of src string from offset to @buf

        define substr(src,buf,offset<#,1>,nchars<#,1>,loc,%done,%err)
        <
         makeaddr(sl,src)
         makeaddr(bl,buf)
         makeaddr(ol,offset)
         makeaddr(nl,nchars)

         move   t2,opsl         ;;assume no immediate mode operand
        ifnb <loc>,<    getlng e,t2>
         getpnt t2,t2           ;;get (length,pointer to 1st byte) of source

;; get byte displacement (offset-1)
         ifl swol,< movei b,opol-1 >
         ife swol,< movei b,-1(opol) >
         ifg swol,< move  b,opol
                    sos   b
                  >
         ifl swnl,< movei a,opnl >
         ifge swnl,< move a,opnl >

;; check if too transfer too big

        ifnb    <loc>,< move    t,b
                        add     t,a
                        camg    t,e ;;compare against length
                        skipge  b
                        jrst    %err
                        >
        adjbp   b,t2            ;;pointer to 1st source byte

;;      get buffer address

        ifl swbl,< movei e,opbl >
        ifge swbl,< move e,opbl >

        hrli    e,^O444400      ;;string pointer to buffer
        move    d,a
        extend  a,[ movslj
                        z ]
        halt    .
        ifnb <loc>,< makeaddr(11,loc)
                     jrst %done
%err:                 .seterr e.soe,op11 ;;Copied Too Far
%done:              >
        >;;end substr

;;      RPLSTR—copies nchar words from @buf to dst string at offset

        define rplstr(buf,dst,offset<#,1>,nchars<#,1>,loc,%done,%err)
        <
         makeaddr(dl,dst)
         makeaddr(bl,buf)
         makeaddr(nl,nchars)
         makeaddr(ol,offset)

         move   t2,opdl
         ifnb <loc>,<getlng     t,t2>
         getpnt t2,t2           ;;get pointer to 1st byte of dst

;; get byte displacement (offset-1)
         ifl swol,< movei e,opol-1 >
         ife swol,< movei e,-1(opol) >
         ifg swol,< move  e,opol
                    sos   e
                  >
         ifl swnl,< movei a,opnl >
         ifge swnl,< move a,opnl >
        ifnb <loc>,< move c,a
                     add  c,e
                     camg c,t   ;;check transfer against string length
                     skipge e
                     jrst %err
                   >
        move    d,a
        adjbp   e,t2

;;      get buffer address

        ifl swbl,< movei b,opbl >
        ifge swbl,< move b,opbl >
        hrli    b,^O444400     ;make full word string ptr.
        extend  a,[MOVSLJ
                       z ]
        halt .
        ifnb <loc>,< makeaddr(ll,loc)
                    jrst %done
%err:                 .seterr e.soe,opll ;;Copied Too Far
%done:              >
        >;;end rplstr

;;
;;INSS -insert string into string

        define inss(src,dst,offset<#,1>,loc,%err)
        <
        makeaddr(sl,src)
        makeaddr(dl,dst)
        makeaddr(ol,offset)

        move    b,opsl
        getlng  a,b
        getpnt  b,b
        ifl swol,< movei e,opol-1 >
        ife swol,<movei  e,-1(opol)>
        ifg swol,<move  e,opol
                  sos   e >
        move    t2,opdl
        getpnt  t2,t2
        ifnb <loc>,<    getlng  t,t2
                        sub     t,e
                        caml    t,a
                        skipge  e
                        jrst    %err
                   >
        adjbp   e,t2
        move    d,a
        extend  a,[movslj
                        z]
        halt    .
        ifnb <loc>,<makeaddr(ll,loc)
                   jrst %done
%err:              .seterr e.soe,opll
```

245

```
%done:          >
        >;;end INSS

;;EXTS -extract new substring from string

        define exts(src,dst,offset<#,1>,nchars<#,1>,loc,%err,%done)
        <
        makeaddr(sl,src)
        makeaddr(dl,dst)
        makeaddr(ol,offset)
        makeaddr(nl,nchars)

        ifl swnl,< movei a,opnl >
        ifge swnl,<move  a,opnl >

        movei   b,strtyp
        movei   c,opdl
        .pushj  p,$$ALLOC##

        ifl swol,< movei b,opol-1 >
        ife swol,<movei  b,-1(opol)>
        ifg swol,<move  b,opol
                  sos   b >
        getpnt  t2,opsl
        ifnb <loc>,<
                     getlng     t,t2
                     sub        t,b
                     sub        t,a
                     skipl      t
                     skipge     b
                     jrst       %err
                >
        adjbp   b,t2
        move    d,a
        getpnt  e,(c)
        extend  a,[movslj
                        z]
        halt    .
        ifnb <loc>,<  makeaddr(ll,loc)
                        jrst %done
%err:                   move    a,(c)
                        .pushj  p,$$DEALC##
                        .seterr e.soe,opll
%done:                    >
        >;;end EXTS
```

```
;;      APPS—appends nchar words of buf(fer) to src string copied to dst

        define apps(buf,src,dst,nchar,loc,%label)
        <
         makeaddr(bl,buf)
         makeaddr(sl,src)
         makeaddr(dl,dst)
         makeaddr(nl,nchar)

         ifge swsl,< move d,opsl >
         ifl swsl,< movei d,opsl >

         %%t2==e
         ifl swnl,< movei e,opnl >
         ife swnl,< %%t2==opnl >
         ifg swnl,< move  e,opnl >

         ifnb <loc>,< makeaddr(ll,loc)
                        jumpge %%t2,%label
                        .seterr e.soe,opll
                    >
        %label:  getlng a,d     ;;length of source string
         movem  d,MARK1##       ;;save adjustable copy of source
         add    a,%%t2          ;;a has length of new string
         movei  b,strtyp
         movei  c,opdl
         .pushj p,$$ALLOC##
         move   t2,e            ;;save nchar
         getpnt b,MARK1
         getlng d,b
         move   a,d             ;;a & d have source string length
         getpnt e,(c)           ;;dest. string pointer
         extend a,[ MOVSLJ
                        z ]
;;make full word byte pointer to buffer
         halt   .

         ifl swbl,< move b,[ ^o444400,,opbl ] >
         ifge swbl,<    hrli    b,^o444400
                        hrr b,opbl >
         move   a,t2
         move   d,t2            ;;a & d have nchars
         extend a,[ MOVSLJ
                        z ]
         halt   .
         setzm  MARK1
        >;;end apps
```

```
>
        end ;lils
```

3.1.4 LILT.MAC

```
        universal LILT — LIL symbol table macros

        search LILDEF

comment /       SYMBOL TABLE BLOCK FORMAT

        ************************************************
        *                       *                      *
WORD 0  *       NAME PTR        *       VALUE PTR      *
        *                       *                      *
        ************************************************
        *                       *                      *
WORD 1  *       FLAG BITS       *       LINK PTR       *
        *                       *                      *
        ************************************************

/

        define lil.t
<       generr(anp,2000)        ;Advance Null Pointer

        define setup(tab,src,dst)      ;sets up call to symbol table subroutines

<       makeaddr(tl,tab)
        makeaddr(sl,src)
        ifnb <dst>,<makeaddr(dl,dst)>

        ifl     swtl,< movei    a,optl >
        ifge    swtl,< move     a,optl >

        ifl     swsl,< movei    b,opsl >
        ifge    swsl,< move     b,opsl >

        .pushj  p,$HASH##
        move    a,t
        ifnb <dst>,<movei       c,opdl>
```

```
>;;end setup

;;      ENTER-- enter source string into symbol table & return address
;;              of symbol table block to destination.
;;              If identical string already entered,return its symbol block
;;              address, & take error jump to loc if loc is given

        define enter(tab,src,dst,loc)
<       setup(<tab>,<src>,<dst>)
        .pushj  p,$ENTER##

        ifb     <loc>,< trn     ;;no-op >

        ifnb    <loc>,< makeaddr(ll,loc)
                        jrst    opll
                ;;      normal return comes here
                        >
>;;end of enter
;;      FIND--search for source string in symbol table. Return symbol table
;;              block to destination ('null' if not found)

        define find(tab,src,dst)
<
        setup(<tab>,<src>,<dst>)
        .pushj  p,$FIND##
        movem   t,opdl
>;;end of find
;;      REMOVE--remove source string from symbol table

        define remove(tab,src,loc)
<       setup(<tab>,<src>,)
        .pushj  p,$REMOVE##
        ifb <loc>,<trn ;;no-op>
        ifnb <loc>,< makeaddr(ll,loc)
                     jrst opll >
>;;end of return
;;      GETVAL--gets value pointer from symbol block

        define getval(src,dst)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

;; (assume source can't be immediate)

        ife     swdl,< hrrz     opdl,@opsl >
        ifn     swdl,< hrrz     t,@opsl
```

```
                        movem   t,opdl >
>

;;      SETVAL--sets value pointer in symbol block

        define setval(src,dst)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

        ifl     swsl,<movei     c,opsl> ;;c gets ptr. to source obj.
        ifge    swsl,< move     c,opsl >

        ;;%%d contains symbol block addr.

        ife     swdl,< %%d==opdl >
        ifg     swdl,< %%d==d
                        move %%d,opdl >

        .pushj  p,HPJECT##      ;;skips if heapject
        jrst    .+5             ;;not a hpject, don´t adjust baclptrs.

        movei   a,NULL
        hlrz    b,-1(c)         ;;get addr. of old owner
        .pushj  p,ADJPTR##      ;;set old owner to NULL
        hrlm    %%d,-1(c)       ;;set backptr. to new owner

        hrrz    a,(%%d) ;;get old value ptr. of symbol block
        .pushj  p,$$DEALC##     ;;deallocate old value
        hrrm    c,(%%d) ;;set new value

>;;end of setval
;;      GETNAM--gets name pointer from symbol block

        define getnam(src,dst)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

;; (assume source can´t be immediate)

        ife     swdl,< hlrz     opdl,@opsl >
        ifn     swdl,< hlrz     t,@opsl
                        movem   t,opdl >
>
;;      GETFLG--get flags from symbol table block

        define getflg(src,dst)
```

```
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

;; (assume source is not immediate)

        ife     swsl,< %%reg==opsl >
        ifg     swsl,< %%reg==t2
                        move    t2,opsl >

        ife     swdl,< hlrz     opdl,1(%%reg) >
        ifg     swdl,< hlrz     t,1(%%reg)
                        movem   t,opdl >
>;;end of getflg

;;      SETFLG—sets flags in symbol table block

        define setflg(src,dst)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

        ife     swdl,< %%reg==opdl >
        ifg     swdl,< %%reg==t2
                        move    t2,opdl >

        ife     swsl,< hrlm     opsl,1(%%reg) >
        ifg     swsl,< hrr      t,opsl
                        hrlm    t,1(%%reg) >
        ifl     swsl,< hrri     t,opsl
                        hrlm    t,1(%%reg) >
>;;end of setflg

;;      GETNXT—advances pointer to symbol table block to next block in chain

        define getnxt(dst,loc,%lab)
        <
         makeaddr(sl,dst)


         hrrz t,opsl
         ifnb <loc>,< makeaddr(ll,loc)
                      caie t,null
                      jrst %lab
                      .seterr(e.amp,opll)
                    >
%lab:    hrrz t,1(t)
         hrrzm t,opsl
        >

>;;end of lil.t

        end
```

3.1.5 LILV.MAC

```
       search lildef

        universal LILV —LIL vector operations


;;
define lil.v
<

        generr(voe,3001)                ; Vector Offset Error

;;      GETOPR— moves an operand in either immediate or memory mode
;;       into a register

        define getopr(reg,opr)
        <
         ifl    sw'opr,< movei  reg,op'opr >
         ifge   sw'opr,< move   reg,op'opr >
        >
;;
;;      SUBVEC —gets nwords of vector pointed to by src starting from offset
;;              words into vector. Places words into block pointed to by dst
;;              Default nwords & offset are 1.

        define subvec(src,dst,offset<#,1>,nwords,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)
         makeaddr(ol,offset)

         ifb <nwords>,<;;default 1 word move

                        move    t,opsl
                        ifl swol,< move t2,<opol-1>(t)>
                        ifge swol,<add  t,opol
                                   move t2,-1(t) >

                        ifl swdl,< movem t2,opdl >
                        ife swdl,< movem t2,(opdl)>
                        ifg swdl,< move  t,opdl
                                   movem t2,(t) >
                        >
        ifnb <nwords>,< makeaddr(nl,nwords)
                        getopr  a,sl
                        getopr  b,dl
                        getopr  c,ol
                        getopr  d,nl
                        .pushj  p,$$SUBV##

        ifnb <loc>,< makeaddr(ll,loc)
                     jumpe      t,%label
                     .seterr    e.voe,opll
%label:
                   >
                        >
        >;;end of .SUBVEC


        define rplvec(src,dst,offset<#,1>,nwords<#,1>,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)
         makeaddr(ol,offset)
         makeaddr(nl,nwords)

         getopr a,dl
         getopr b,sl
         getopr c,ol
         getopr d,nl
         .pushj p,$$RPLV##

        ifnb <loc>,< makeaddr(ll,loc)
                     jumpe t,%label
                     .seterr e.voe,opll
        %label:
                   >
        >
;;
;;INSN — inserts number vector in number vector at offset (default 1)

        define insn(src,dst,offset<#,1>,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)
         makeaddr(ol,offset)

         getopr a,sl            ;;a gets ptr. to source vector
         getopr b,dl
         getopr c,ol
         getlng d,a             ;;size of transfer is source length
         .pushj p,$$SUBV##

         ifnb <loc>,<   makeaddr(ll,loc)
                        jumpe   t,%label
                        .seterr e.voe,opll
%label:
                    >
        >;;end of INSN

;;EXTN --extract new number vector from number vector

        define extn(src,dst,offset<#,1>,nwords<#,1>,loc,%label)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)
         makeaddr(ol,offset)
         makeaddr(nl,nwords)

         getopr d,sl
         movem  d,MARK1##
         getopr a,nl            ;;length of new vector
         movei  b,numtyp
         movei  c,opdl
        .pushj  p,$$ALLOC##
         move   d,a
         move   b,(c)
         getopr c,ol
         move   a,MARK1
         setzm  MARK1
        .pushj  p,$$SUBV##
        ifnb <loc>,     <makeaddr(ll,loc)
                         jumpe  t,%label
                         move   a,opdl
                         .pushj p,$$DEALC##
                         .seterr        e.voe,opll
%label:                 >
        >;;end of EXTN

>;end lil.v
```

```
    end
```

3.1.6 LIL.MAC

```
    universal LIL
     ; LIL macro generator for the PDP-10

     search LILDEF ;contains global definitions for PDP-10 LIL


     subttl MAKEADDR—generates pdp-10 addresses from LIL adress pairs


     ; returns type of addressing mode (register,immediate,memory)
     ; by setting created symbol  "sw'type'"

     ; makeaddr returns  the pdp-10 address
     ; specified by the lil address pair (addr1,addr2)
     ; if addr1='#' then immediate mode so return addr2
     ; if addr1='@' then indirect register so return (addr2)
     ; else addr1,addr2 are address,register pair so return addr1(addr2)
     ; (if addr2 is missing,it defaults to 0,but 0 is not an index register)
     ;note all registers are offset by 'lilreg'

     define makeaddr(type,addr1,addr2<-lilreg>)
     < define op'type
      <addr1(addr2+lilreg)>
       %%mem==0
       sw'type==memor

      ifidn <addr1><#>,< %%mem==1
                         %%t==0  ;;zero %%t means value available for addr2

                        .ifn addr2,numeric,<
                         .if addr2,extern,   <%%t==1>
                         ifndef addr2,       <%%t==1>
                                             >
                         ifg %%t,< define op'type < [ addr2 ] >>

                         ife %%t,<
                                 ifl addr2,< define op'type
                                             < [ addr2 ] >
```

```
                                                 >
                                    ifge addr2,< sw´type==immed
                                                 define op´type
                                                  <addr2 >
                                                 >
                                    ifg addr2-^o777777,<define op´type
                                                <  [addr2] >
                                                 sw´type==memor >
                                    >
                          >
        ifidn <addr1><@>,< sw´type==indir
                           %%mem==1
                           define op´type
                            <(addr2+lilreg)>>

;;check if memory reference is really a register

        ife %%mem,< ifge addr2,<%%mem==1> >
        ife %%mem,<  .if addr1,absolute,<
                       ifge <addr1>,<
                        ifle <<addr1>-17>,< sw´type==regis
                                            define op´type<addr1>
                                           >

                                     >
                                        >

                  >
        >
        subttl APR trapping macros

        define settrap(loc< >)
        ;loc is address of routine to jump to on APR
        ;if loc is null,$aprjp (trap location word in LILENV) is zero
        ;SETTRAP generates its own inverse ,UNTRAP
        <
         ifb <loc>,<    define untrap
                           < >
                 >

        ifnb <loc>,<    makeaddr(11,<loc>)

                        movei t,op11
                        movem t,$aprjp##

                        define untrap
                        < setzm $aprjp >
                >
        >
        subttl LIL.2
        ;LIL.2 generates all 2 operand,commutative,non-string lil instructions

        define lil.2(lilop,suffix,opcode)
        ;lilop--basic lil instruction name
        ;suffix--data type suffix(i.e. C,I,P (but not S))
        ;opcode--pdp-10 opcode
        <


                define .'lilop(src,dst,loc)
                < makeaddr(sl,src)
                  makeaddr(dl,dst)
                  settrap(loc)
                 ifl swsl,
                        < ;; src is immediate
                         ife swdl,< .'opcode'i opdl,opsl >
                         ifg swdl,< movei t,opsl
                                    opcode'm t,opdl >
                        >
                ife swsl,
                        < ;;src is register
                          opcode'm opsl,opdl >
                ifg swsl,
                        < ;;src is memory reference
                         ife swdl,< opcode opdl,opsl >
                         ifg swdl,< move t,opsl
                                    opcode'm t,opdl >
                        >
                 untrap
                > ;;end of lilop

        ;;now generate synonyms with suffixes

        irp suffix, <   syn .'lilop,lilop'suffix
                    >

> ;end of LIL.2

;       SUBm

        define .sub(src,dst,loc)
<
        makeaddr(sl,src)
        makeaddr(dl,dst)

        settrap(loc)
        ife     swdl,<
                        ifl swsl,< .subi opdl,opsl >
                        ifge swsl,< sub  opdl,opsl >
                     >
        ifg     swdl,<
                        ifl swsl,< move t,opdl
                                   .subi t,opsl
                                   movem t,opdl
                                 >
                        ifge swsl,< move t,opdl
                                   sub  t,opsl
                                   movem t,opdl
                                  >
                     >
>;end .subi
        SYN .sub,subi
        SYN .sub,subc
        SYN .sub,subp

        subttl REMI

        define remi(src,dst,loc)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)

         settrap(loc)

         move   t,opdl
         ifl swsl,< idivi t,opsl>
         ifge swsl,<idiv  t,opsl>
;; now remainder is in t2
         movem t2,opdl

        untrap

        >

        subttl QUOI

        define quoi(src,dst,loc)
        <
         makeaddr(sl,src)
         makeaddr(dl,dst)

         settrap(loc)

         move   t,opdl
```

```
  ifl swsl,< idivi t,opsl>
  ifge swsl,<idiv  t,opsl>

  movem t,opdl

 untrap

 >

 define ldivi(src,dstl,dst2,loc) ;;long divide
 <
  makeaddr(sl,src)
  makeaddr(dl,dstl)
  makeaddr(d2,dst2)

  settrap(loc)

  move   t,opdl
  move   t2,opd2
  ifl swsl,< divi t,opsl>
  ifge swsl,<div  t,opsl>

  movem t,opd2
  movem t2,opdl

 untrap

 >

 define udivi(src,dstl,dst2,loc) ;;"undivide" operation
 <                                ;;double length result
  makeaddr(sl,src)
  makeaddr(dl,dstl)
  makeaddr(d2,dst2)

  settrap(loc)

  ifl swsl,< movei t,opsl>
  ifge swsl,<move  t,opsl>

  mul    t,opdl
  setz   a,
  move   b,opd2
  dadd   t,a
  movem  t2,opdl
  movem  t,opd2

 untrap
```

```
>

define  laddi(src,dst,loc)
<
 makeaddr(sl,src)
 makeaddr(dl,dst)
 makeaddr(ll,loc)

 setzb  t,a
 move   t2,opdl
 ifl swsl,< movei b,opsl >
 ifge swsl,< move b,opsl >
 dadd   t,a
 movem  t2,opdl
 jumpn  t,opll
>

subttl Swapp —one owner move pointer

define swapp(src,dst)
<
 makeaddr(sl,src)
 makeaddr(dl,dst)

 ifl    swsl,< movei    b,opsl >
 ifge   swsl,< move     b,opsl >

 movei  a,opdl
 .pushj p,$SWAPP##
>


define omovp(src,dst);; move ownership without deallocating dst
<
 makeaddr(sl,src)
 makeaddr(dl,dst)

 ifl    swsl,< movei    c,opsl >
 ifge   swsl,< move     c,opsl >

 move   a,c
 .pushj p,HPJECT##
 jrst   .+6             ;;skip if not heap object

;; else adjust backpointer of source
        hlrz   b,-1(c)          ;;get backpointer
        movei  a,NULL           ;;former owner gets NULL
 .pushj p,ADJPTR##
 movei  a,opdl
 hrlm   a,-1(c)          ;;set new backpointer

 movem  c,opdl
>

subttl LIL.1—1 operand,non-string lil macros

define lil.1(lilop,suffix,opcode)
;lilop—basic lil instruction name
;suffix—data type suffix(i.e. C,I,P (but not S))
;opcode—pdp-10 opcode
<
 define .'lilop(dst)
 <
  makeaddr(dl,dst)

  opcode opdl
 >

;;generate synonyms

irp suffix,<syn .'lilop,lilop'suffix
          >

>;end of LIL.1


subttl LIL.1J—1 operand jump macros

define lil.1j(suffix,cond,cmpcnd,typsuf)
;cond—suffix for PDP-10 JUMP or SKIP instruction
;cmpcnd—complement of cond
;suffix—suffix for LIL JMP instruction
;typsuf—suffix for type of LIL args
<
        define jmp'suffix(src,loc)
        <
         makeaddr(sl,src)
         makeaddr(ll,loc)

         ifl swsl, ;;source is immediate
                < movei t,opsl
                  jump'cond t,opll >

         ife swsl, ;;source is register
                < jump'cond opsl,opll >
          ifg sws1, ;;source is memory
                    ip´cmpcnd ops1
                  jrst opll >

         > ;;end jmp´suffix

 irp typsuf, < syn jmp´suffix, jmp´suffix´typsuf
          >
> ;end LIL.1J


;JMP     t created by lil.1j
define jmp(loc)

  akeaddr(ll,loc)

 jrst opll
> ;end jmp

subttl LIL.2J--2 operand compare & jump macros

define lil.2j(suffix,cond,rcond,typsuf)
;cond---suffix for PDP-10 compare instruction,complement of LIL suffix
;rcond--suffix for PDP-10 compare instruction when second operand is
;       a register
;suffix--LIL JMP instruction suffix
;typs r--LIL arg type suffix
<
        define jmp´suffix(src1,src2,loc)
        <
         makeaddr(s1,src1)
         makeaddr(s2,src2)
         makeaddr(ll,loc)

         ifl sws1,      ;;src1 is immediate
                <
                 ifl sws2,      ;;src2 is immediate
                        < movei t,ops1
                          cai´cond t,ops2 >

                 ife sws2,      ;;src2 is register
                        < cai´rcond ops2,ops1 >

                 ifg sws2,      ;;src2 is memory
                        < movei t,ops1
                          cam´cond t,ops2 >
                >
          ife swsl,      ;;srcl is register
                <
                 ifl sws2,      ;;src2 is immediate
                       < cai´cond opsl,ops2 >

                 ifge sws2,     ;;src2 is register or memory
                       < cam´cond opsl,ops2 >
                >

          ifg swsl,      ;;srcl is memory
                <
                 ifl sws2,      ;;src2 is immediate
                       < move t,opsl
                         cai´cond t,ops2 >

                 ife sws2,      ;;src2 is register
                       < cam´rcond ops2,opsl >

                 ifg sws2,      ;;src2 is memory
                       < move t,opsl
                         cam´cond t,ops2 >
                >

         jrst opll ;;jump if condition satisfied(no skip)

         > ;;end jmp´suffix

  irp typsuf, < syn jmp´suffix, jmp´suffix´typsuf
            >
 > ;end LIL.2J

 subttl STACK macros

 define pop.p(suffix)
 ;suffix--list of LIL  type suffixes
 <
         define pop(oprnd)
         <
          makeaddr(ol,oprnd)

          .pop p,opol

         > ;;end of pop

  ;;generate synonyms

  irp suffix,< syn pop, pop´suffix
```

263

```
            >

> ;end of pop.p

define push.p(suffix)
;suffix--list of LIL type suffixes
<
        define push(oprnd)
        <
         makeaddr(ol,oprnd)

         ifl swol,               ;;oprnd is immediate
                <
                 movei t,opol
                 .push p,t
                >

         ifge swol,              ;;oprnd is reg. or mem.
                <
                 .push p,opol
                >
        > ;;end of push

 ;; generate synonyms

 irp suffix, < syn push, push'suffix
             >
> ;end of push.p


define pushm(reg1,reg2,%p)
;pushes LIL registers from reg1 to reg2
<
 makeaddr(r1,reg1)
 makeaddr(r2,reg2)

 %p==opr1
 repeat <opr2>-<opr1>+1,
        <
         .push p,%p
         %p==%p+1
        >
> ;end of pushm

define popm(reg1,reg2,%p)
;inverse of pushm
<
```

```
         makeaddr(r1,reg1)
         makeaddr(r2,reg2)

         %p==opr2
         repeat <opr2>-<opr1>+1,
                <
                 .pop p,%p
                 %p==%p-1
                >
        > ;end of popm

        define call(loc1,loc2)
        <
         makeaddr(l1,loc1)

         pushj p,opl1

         ifnb <loc2>,< makeaddr(l2,loc2)
                        jrst    opl2    ;;failure return
                     >
>;;end call

        SYN call,pcall                  ;;no special implementation of PCALL

        define return
        <
         popj p,
        >

        SYN return,fail                 ;failure is ordinary return

        define succeed
        <
         aos    (p)
         .popj  p,                      ;;success is skip return
        >
        subttl HOST--calls host (LILENV) subroutine

        define host(loc)
        < .pushj p,loc´## >

        subttl Flag Macros

        define tstb(flag,src,loc)
        <
         makeaddr(f1,flag)
         makeaddr(s1,src)
         makeaddr(l1,loc)
```

```
 ifl swsl,<setcmi t,opsl >
 ifge swsl,<setcm t,opsl >

 ifl swfl,< trnn t,opfl >
 ifge swfl,<tdnn t,opfl >
  jrst opll
>

define setb(flag,src)
<
 makeaddr(fl,flag)
 makeaddr(sl,src)

 ife swsl,<%%t==opsl >
 ifg swsl,<%%t==t
           move t,opsl >

 ifl swfl,< tro %%t,opfl >
 ifge swfl,<tdo %%t,opfl >

 ifg swsl,< movem %%t,opsl >
>
define clrb(flag,src)
<
 makeaddr(fl,flag)
 makeaddr(sl,src)

 ife swsl,<%%t==opsl >
 ifg swsl,<%%t==t
           move t,opsl >

 ifl swfl,< trz %%t,opfl >
 ifge swfl,<tdz %%t,opfl >

 ifg swsl,< movem %%t,opsl >
>

subttl ERROR macros

define seterr(code,loc)
<
 makeaddr(cl,code)
 makeaddr(ll,loc)

 ifl    swcl,<movei     t,opcl
              movem     t,$$ERCD##>
 ifg    swcl,<move      t,opcl
```

```
         movem  t,$$ERCD##>
 ife    swcl,<movem opcl,$$ERCD##>

 movei  t,.+3
 movem  t,$$ERAD##
 jrst   opll

>

define geterr(code,loc)
<
 makeaddr(cl,code)
 makeaddr(ll,loc)

 ifg swcl,< move t,$$ERCD##
            movem t,opcl>
 ife    swcl,< move opcl,$$ERCD##>

 ifg swll,< move t,$$ERAD##
            movem t,opll>
 ife    swll,< move opll,$$ERAD##>
>

subttl Heap object macros

define getlen(src,dst)
<
 makeaddr(sl,src)
 makeaddr(dl,dst)

 ife    swsl,<
              ife       swdl,< hrrz opdl,-2(opsl) >
              ifg       swdl,< hrrz    t,-2(opsl)
                                movem  t,opdl
                              >
             >
 ifg    swsl,<
              move      t,opsl
              ife       swdl,< hrrz     opdl,-2(t) >
              ifg       swdl,< hrrz       t2,-2(t)
                                movem     t2,opdl
                              >
             >
>;end getlng

define gettyp(src,dst)
<
 makeaddr(sl,src)
```

267

```
 makeaddr(dl,dst)

 ife    swsl,<
             ife       swdl,< hlrz opdl,-2(opsl) >
             ifg       swdl,< hlrz    t,-2(opsl)
                                movem  t,opdl
                             >
             >
 ifg    swsl,<
             move      t,opsl
             ife       swdl,< hlrz      opdl,-2(t) >
             ifg       swdl,< hlrz        t2,-2(t)
                                movem     t2,opdl
                             >

             >
>;end gettyp

define settyp(src,dst)
<
 makeaddr(sl,src)
 makeaddr(dl,dst)

 ifl    swsl,< movei    t,opsl >
 ifge   swsl,< move     t,opsl >

 ife    swdl,< hrlm     t,-2(opdl) >
 ifg    swdl,< move    t2,opdl
                hrlm    t,-2(t2)
             >
>;end settyp

subttl LIL psuedo-ops

define psect
< ife $sflag-1,< $vsect=. >
  reloc $psect
  $sflag==0
>
define vsect
< ife $sflag,< $psect=. >
  reloc $vsect
  $sflag==1
>
define const(name,value)
< name=value
 >

define include(filnam)
```

```
< search filnam         ;;not really it,but the best macro-10 can do
>

define extrn
< external >

define module(name,switch)
<
 .require LILDAT        ;;loads run-time data area
 .require LILLIB        ;;loads run-time environment
 define prgnam
 < ifidn <switch><include>,<universal name >
   ifb   <switch>,<title name>
 >
 if2 < printx MODULE name >
 search LILDEF,MONSYM
 radix 10
 ifb <switch>,<
               twoseg $psini
                reloc  $psini
                $psect=$psini
                $vsect=0                  ;;start of vsect
                $sflag==0                 ;;0--in psect,1 means in vsect,2 --dsect
             >
>
define start(addr)
< $start=:addr >

define endmod(name)
< if2 < printx end MODULE name >
  prgnam
  end
>

define sbttl(string)
< page
  subttl string
>

define ptr
<exp >

define int
<exp >

define text(arg)
<.shift==^d36-bytsiz
 %flag==0
```

269

```
 %stop==0
 irpc   arg,<
               ifn %stop,<STOPI>
               ifn %flag,<
                ifidn <arg><">,<
                                  ifn %shift-.shift,<exp %word>
                                  %stop==1
                                >
                ifdif <arg><">,<
                 %word==%word+<ascii"arg">_<%shift-^d29>
                 %shift==%shift-^d9
                 ifl %shift,< exp %word
                                  %word==0
                                  %shift==.shift
                                  >
                                >
                           >
               ife %flag,< ifidn <arg><">,<%flag==1
                                            %word==0
                                            %shift==.shift
                                           >
                          >
               >
>

define blkp
< block >

define blki
< block >

define prim(procname,flags,procaddr,syment)
< ifnb <procaddr>,< external procaddr >
  ifnb <syment>,  < external syment   >
  external procname
  exp procname, flags,  procaddr, syment>

define pfix(type,size,flags<0>)
<
      ife       type-vectyp,<%%t==size>
      ife       type-strtyp,<%%t==<size+bytpw-l>/bytpw
                            >
      xwd type+flags,size
      xwd       0,%%t
>


subttl generating section
```

```
lil.1(inc,<i,c,p>,aos)
lil.1(dec,<i,c,p>,sos)

;JMP is defined separately
lil.1j(z,e,n,<i,c,p>)
lil.1j(nz,n,e,<i,c,p>)
lil.1j(p,g,le,<i,c,p>)
lil.1j(np,le,g,<i,c,p>)
lil.1j(m,l,ge,<i,c,p>)
lil.1j(nm,ge,l,<i,c,p>)

lil.2j(l,ge,le,<i,c,p>)
lil.2j(e,n,n,<i,c,p>)
lil.2j(ne,e,e,<i,c,p>)
lil.2j(g,le,ge,<i,c,p>)
lil.2j(nl,l,g,<i,c,p>)
lil.2j(ng,g,l,<i,c,p>)

lil.2(mov,<i,c,p>,move)
lil.2(add,<i,c,p>,add)
lil.2(mul,i,imul)
lil.2(and,<i,c,p>,and)
lil.2(ior,<i,c,p>,ior)
lil.2(xor,<i,c,p>,xor)
; remi,quoi & sub(m) defined separately

push.p(<i,c,p>)
pop.p(<i,c,p>)


search LILS ;string macros
lil.s

search LILC ;conversion macros
lil.c

search LILT ;symbol table macros
lil.t

search LILV ;vector handling macros
lil.v

;now purge generating macros to save UNV space

purge lil.1,lil.1t,lil.1j,lil.2,lil.2j,push.p,pop.p,lil.s,lil.c,lil.t
purge lil.v
end ;of LIL
```

3.2 The Host Environment

All the files listed in this section produce read-only code, with the exception of LILDAT.MAC which contains data only. The output of the assembled files are TOPS-20 relocatable binary files ("REL" files). These object files should, with the exception of LILDAT.REL, be combined into a single file called LILLIB.REL. (The TOPS-20 MAKLIB utility can be used for this purpose).

3.2.1 LILA2T.MAC

```
        title LILA2T -- add integer to string pointer routine(KL only)

         search lildef

         twoseg  $psini
         reloc   $psini

;        ROUTINE $ADD2T,ARG a-b,RESULT t
;        adds byte displacement in a to string pointer in b

         entry   $ADD2T

$ADD2T: move    t,a
         adjbp   t,b
         popj    p,

         End
```

3.2.2 LILALC.MAC

```
comment/
         This is a hand-coding of the PASCAL allocator (ALLOC.PAS).
         Some routines called only once have been placed in-line
         /
```

```
comment/  ( 21 November 1979 by RJS)

        LIL macros MOVS,CATS & APPS can mark transient pointers to
        be updated if compaction occurs during the execution of the macro
        /
comment/  ( 23 January 1980 by RJS)

        Simplify allocator by eliminating free list. Free objects are marked
        by the bit %FREE being turned on
        /
comment/  Friday, 23 May 1980, RJS

        add trapping of store overflows instead of bombing out
        /

        title   LILALC--LIL, storage allocator interface

        search lildef,monsym, CONSTS

        twoseg  $psini

comment/        ALLOCATOR DATA STRUCTURES

1)  Heap Object Prefix

                0           15  16  17  18
        ^       ************************************************
   P    !       *           * n * v * s *                      *
   F    !       *   FLAGS   * u * e * t *    LOGICAL SIZE      *
   X    !       *           * m * c * r *                      *
   S    !       ************************************************
   I    !       *                       *                      *
   Z    !       *       BACKPOINTER     *    PHYSICAL SIZE     *
        !       *       ( to owner )    *(no. of words of data) *
        v       ************************************************
                *                                              *
                *                  N I N T S                   *
                *       (no. of integers in vectors only)      *
                ************************************************

2) Headers of Free List Objects

                 0(%FREE)
                ************************************************
                * *                     *                      *
                *1*        0            *    FREE LENGTH       *
                * *                     *                      *
```

273

****

3) Name Table Entries

(see LILT.MAC)

```
/
        subttl global variables & symbols (in LILDAT)

        external HPBOT, HPTOP, STFREE, BIGSIZ, FRENAM, NMBOT, LASTR4
        external MARK1,MARK2     ;change lev. 1

        subttl  Allocator predicates & utilities
;
; all predicates signal true by a skip return


        reloc   $psini

; HRJECT -- true if addr. in c is in heap

        entry HRJECT             ; (also called by SETVAL macro)

HRJECT: camge   c,HPBOT
        popj    p,
        camge   c,HPTOP
        aos     (p)
        popj    p,

; NMJECT -- true if addr. in b is in name space

NMJECT: camge   c,NMBOT
        popj    p,
        camge   c,HPBOT
        aos     (p)
        popj    p,

; LENGTH -- returns physical length of an object whose logical length
;         is in a and type is in b

LENGTH: move    t,a
        trnn    b,STRTYP
        popj    p,
        idivi   t,BYTPW          ; bytes per word, t2 has excess bytes
        skipe   t2
        aos     t
        popj    p,

; ADJPTR -- adjusts an owner of a heap object
```

```
;input regs:
;               a — new value for owner
;               b — address of owner
;               c — address of heapobject

;
; look for object first in right, then left half of owner

        entry ADJPTR            ; (also called by SETVAL macro)

ADJPTR: hrrz    t,(b)
        came    t,c
        jrst    ADJLEF
        hrrm    a,(b)
        popj    p,

ADJLEF: hlrz    t,(b)
        came    t,c
        jrst    ADJERR  ; it's got to be in left or right half
        hrlm    a,(b)
        popj    p,

ADJERR: hrroi   a,[asciz/Ownership discrepancy--/]
        psout
        jrst    GIVEUP##

EXPAND: hrroi   a,[asciz/NAME TABLE FULL
/]
        psout
        pushj   p,$COMPACT      ;eliminate deleted heapjects for NOSTORE
        jrst    NOSTORE

        subttl HPSTRT--initializes heap. Called from LILENV.

        entry HPSTRT

HPSTRT: move    d,.JBREL##
        aoj     d,              ;start name table just above loaded low segment
        movem   d,NMBOT
        movei   b,NTABSZ(d)
        movem   b,HPBOT
        movem   b,HPTOP
        setzm   (b)
        movei   t,STACK
        movem   t,LASTR4        ;STACK will be initial value for R4
        setzm   STFREE          ;i.e., initially no free objects
;
```

```
; Initialize name table free list

        movei   t,NULL
HPST1:  movem   t,1(d)
        move    t,d
        addi    d,2
        caige   d,-1(b)
        jrst    HPST1
        movem   t,FRENAM
        popj    p,              ; return

        subttl COMPACTor

comment!

 compacts storage & unmaps free pages from process

 SHIFT is coded in-line

 register usage:

        a — HOLESTART ( lowest free address )
        b — addr. of owner of a shifted object
        c — new addr. of object, then object size (OBJSIZE)
        d — next object in heap
!
        entry   $COMPACT

$COMPACT:

        push    p,a
        push    p,b
        push    p,c
        push    p,d
;       hrroi   a,[asciz/Compaction.../]
;       psout
;       movni   a,5
;       runtm
;       movem   a,COMRNT##      ;store runtime so far

        skipg   a,STFREE        ; get addr. of first hole in heap
        jrst    COMEND          ; do nothing if no free objects(STFREE is 0)
        hlrz    d,1(a)
        addl    d,a             ;d has addr. of next obj.
;
; main compaction loop

COMPA2: caml    d,HPTOP         ; exit after going thru entire heap
```

```
        jrst    COMPA5

        move    t,(d)           ;get 1st prefix word of heapject
        tlzn    t,%FREE         ; if NEXTOBJ is not on
        jrst    SHIFT           ; free list, shift, else just move to next object
        add     d,t             ; calc. addr. of next heapject( t now contains
        jrst    COMPA2          ; size of free object
;
; SHIFT coded in-line with a BLT used to transfer prefix & data

SHIFT:  hlrz    b,1(d)          ; get addr. of owner
        movei   c,pfxsiz(d)     ; addr. of object
        addi    a,pfxsiz        ; a becomes new addr. of data
        pushj   p,ADJPTR        ; change owner
;
;change level 1--add adjusting of marked temporaries

        camn    c,MARK1         ;check if shifted object is marked
        movem   a,MARK1         ;adjust

        camn    c,MARK2         ;check if shifted object is marked
        movem   a,MARK2         ;adjust
;
;end of change

        subi    a,pfxsiz        ; restore a

        hrl     t,d             ; start of old object
        hrr     t,a             ; start of new object
        hrrz    c,1(d)          ; OBJSIZE
        move    t2,c
        addi    t2,-1(a)        ; last addr. in new object
        blt     t,(t2)
;
; adjust backpointers for pointers within vectors

        hlrz    t,(a)           ; get flags
        trnn    t,vectyp
        jrst    COMPA4

        move    t,pfxsiz(a)     ; get NINTS
        addi    t,pfxsiz(a)     ; addr. of 1st ptr. in vector
COMPA3: camle   t,t2
        jrst    COMPA4
        move    b,(t)           ; get addr. of object referenced by ptr.
        camge   b,HPBOT         ; in-line coding of HPJECT
        aoja    t,COMPA3
        camge   b,HPTOP
        hrlm    t,-1(b)         ; change backpointer for heapjects only
        aoja    t,COMPA3

COMPA4: add     d,c
        add     a,c
        jrst    COMPA2          ; end of main loop
;
; reset allocator globals & release now unused pages

COMPA5: setzm   STFREE          ;indicate no free objects
        movem   a,HPTOP
        move    b,a
        ash     b,-9            ; get page no. of heap top
        addi    b,1             ; b has first free page
        move    c,(r4)
        ash     c,-9            ; page no. of top of stack
        sub     c,b             ; no. of pages to release
        jumple  c,COMEND
        tlo     c,400000        ; multiple page PMAP
        hrli    b,400000        ; this process
        seto    a,              ; unmap pages
        pmap

COMEND: move    t,(r4)          ; get addr. of top of stack
        sub     t,HPTOP
        movem   t,BIGSIZ        ; also total amt. of free storage

;       movni   a,5             ;entire job
;       runtm
;       sub     a,COMRNT
;       move    b,a
;       movei   c,^d10
;       movei   a,.priou
;       nout
;       halt    .
;       hrroi   a,[asciz/ ms run time
;/]
;       psout

        pop     p,d
        pop     p,c
        pop     p,b
        pop     p,a
        popj    p,

        subttl $ALLOC

;       $ALLOC is called by LIL ´host´ macro with args in LIL registers
;       r1—size of object
;       r2—type
;       r3—addr. of pointer

        entry $ALLOC

$ALLOC: move    t,[ r1,,a ]
        blt     t,c               ; load a-c from r1 - r3
        jrst    $$ALLOC           ; call Allocator
                                  ; return from there

        subttl  ALCFRE — allocates new object

comment/
         input regs:

                a — logical size of new object
                b — type
                c — ptr. to object's owner
                d — physical size of object
                e — addr. of prefix of new object
/
ALCFRE: hrlm    b,(e)             ; set up prefix
        hrrm    a.(e)
        hrlm    c,1(e)
        hrrm    d,1(e)

        movei   t,pfxsiz(e)       ; set owner
        movem   t,(c)

        trnn    b,vectyp          ; if vector, make NINTS 1
        popj    p,                ; & fill rest with NULLs
        movei   t,1
        movem   t,pfxsiz(e)
        caig    d,3               ;return if object size is 1
        popj    p,
        movei   t2,pfxsiz+1(e)    ;addr. of start of rest of vector
        movei   t,NULL
        movem   t,(t2)
        caig    d,4
        popj    p,
        hrls    t2
        aos     t2
        movei   t,-1(e)
        add     t,d               ;t is last addr. in vector
        blt     t2,(t)
        popj    p,
        subttl  $$ALLOC--calls main storage allocator from LIL macros

        ;addr. of pointer to object in c
        ;receives size & type of new object in a,b
        ;allocates new object in heap with correct prefix

        entry $$ALLOC

$$ALLOC:push    p,d
        push    p,e

        move    e,a             ; store a temporarily
        move    a,(c)
        pushj   p,$$DEALC       ; deallocate old object first
        move    a,e             ; restore a

        jumpe   a,ALCEMP        ; return EMPTY for 0 length objects
        pushj   p,LENGTH        ; physical length of object in t
        movei   d,pfxsiz(t)     ; d will be PHYSSIZE

        move    e,(r4)          ; get TOPOFSTACK
        sub     e,hptop
        movem   e,BIGSIZ        : (BIGOBJSIZE)
        caig    e,DANGER
        jrst    NOFREE          ; go compact if heap too close to stack

        caml    e,d             ;Go allocate if room for new object on top of heap
        jrst    ALLOC1          ;  else compact
;
; no free object big enough, compact & try again

NOFREE: pushj   p,$COMPACT
        move    e,BIGSIZ        ;(BIGSIZ updated by $COMPACT)
        caile   d,-DANGER(e)    ; enough space on top of heap?
        jrst    NOSTORE         ;bomb if < DANGER would remain after allocation
        caile   e,WARN          ;set $PI if too little heap space
        jrst    ALLOC1
        movei   t,T$ALC         ;(from CONSTS)
        movem   t,$PI##
;
; Object allocated from top of heap

ALLOC1: move    e,hptop
        pushj   p,ALCFRE        ; call ALLOCFREE (a-e unchanged)
        addm    d,hptop         ; add PHYSSIZE to HEAPTOP
        jrst    ALCRET          ; & return
;
; return empty object
```

```
ALCEMP: movei   t,EMPTY##
        movem   t,(c)

ALCRET: pop     p,e
        pop     p,d
        popj    p,

        subttl NOSTORE
;
;not enough free storage.  Reset system stack.
;Reenter LIL thru $BOMB (LGOINI)

NOSTORE:
        movei   t,T$ALC
        movem   t,$PI
        move    p,[iowd stksiz-1,$stack+1 ] ;leave former top element on stack
                                    ; for return to LILENV
;
;for now, delete all heapjects owned by pointers between heap & stack
;                           or owned by a (LIL) register
;(these are objects which can't be deallocated via SWAPPs because
; we can only tell if they (the owners) contain integers or pointers
; by scanning the heap)

        move    e,HPBOT         ;e will point to current heapject
NSTOR2: caml    e,HPTOP         ;end of heap?
        jrst    $BOMB##         ;yes, re-enter LOGO (in LGOINI)
        hlrz    t,1(e)          ;get backpointer to owner

        caig    t,R4            ;is owner register?
        jrst    NSTOR3          ;yes

        camge   t,HPTOP
        jrst    NSTOR4
        caml    t,(R4)
        jrst    NSTOR4

NSTOR3: movei   a,pfxsiz(e)     ;adjust addr. for $$DEALC
        pushj   p,$$DEALC

NSTOR4: hrrz    t,1(e)          ;get physical length
        add     e,t             ;calc. addr. of next heapject
        jrst    NSTOR2

        subttl $DEALC --calls allocator for deallocation
```

281

```
;       explictly called from LIL code
;       addr. of pointer to de-allocated object in LIL register 1

        entry $DEALC

$DEALC: move    a,(rl)          ; addr. of object in a
        jrst    $$DEALC

        subttl $$DEALC—deallocator called from LIL expansion or from
;                           inside LILALC

;       addr. of object in a

        entry $$DEALC

$$DEALC:push    p,c

        move    c,a             ; c has addr. of object to be deallocated
        pushj   p,HPJECT
        jrst    DERET2          ; if not in heap, just return

        push    p,b
        push    p,d
;
; if object is vector, first recursively deallocate all ptrs. in vector

        hlrz    t,-2(c)         ; get flags
        trnn    t,vectyp
        jrst    DEALC1          ; jump if not vector

        hrrz    b,-2(c)         ; logical length of vector
        move    d,(c)           ; NINTS in d
        sub     b,d             ; no. of ptrs. in vector
        jumple  b,DEALC1
        add     d,c             ; d now points to 1st pointer in vector

DVECT:  move    a,(d)
        pushj   p,$$DEALC       ; deallocate pointer
        aoj     d,              ; get next pointer
        sojg    b,DVECT
;
; change backpointers & mark free object

DEALC1: movei   a,NULL          ; a used as temporary
        hlrz    b,-1(c)         ; b is owner
        pushj   p,ADJPTR        ; make owner NULL

        hrrz    t,-1(c)         ;get physical length
        move    t2,HPTOP
        sub     t2,t            ;check if object is at top of heap
        caie    t2,-pfxsiz(c)
        jrst    DEALC2
        movem   t2,HPTOP        ;yes, adjust top of heap & return
        jrst    DEARET

DEALC2: hrli    t,%FREE
        movem   t,-2(c)         ;place in prefix

        movei   t,-2(c)         ;addr. of new free object
        skipe   STFREE          ;if no free objects, always adjust STFREE
        camge   t,STFREE
        movem   t,STFREE        ;adjust STFREE to be lowest free object
;
; return from deallocator

DEARET: pop     p,d
        pop     p,b
DERET2: pop     p,c
        popj    p,




        Subttl $ALCBLK
;       allocates two word block for symbol table usage
;       c contains addr. of pointer to new block

        entry $ALCBLK

$ALCBLK:
        move    t,FRENAM
        cain    t,NULL          ; if no free names,
        pushj   p,EXPAND        ; try to expand name area(currently fails)
        movem   t,(c)
        hrrz    t2,1(t)         ; get next block on free name list
        movem   t2,FRENAM
        popj    p,

        Subttl $SWAPP
```

283

```
;       does SWAPP operation—preserves regs.

; input regs—(only the right halves are used)
;
;       a—addr. of destination ptr.
;       b—addr. of source obj.

        entry   $SWAPP

$SWAPP: push    p,a
        push    p,b
        push    p,c
        push    p,d

        hrrz    d,a             ; store dest. ptr. in d
        hrrz    c,b             ; store addr. of source in c

        move    a,(d)           ; deallocate old contents of destination
        pushj   p,$$DEALC
        move    a,c             ; get addr. of source object in a
        pushj   p,HPJECT
        jrst    SWAPP1
;
; if source is heapject, place NULL in former owner

        hlrz    b,-1(c)         ; get owner of source
        movei   a,NULL
        pushj   p,ADJPTR
        hrlm    d,-1(c)         ; adjust backpointer
;
; move source data to destination

SWAPP1: movem   c,(d)

        pop     p,d
        pop     p,c
        pop     p,b
        pop     p,a
        popj    p,

        end
```

3.2.3 LIILCMP.MAC

```
       title LIILCMP--compares two strings alphanumerically

        search lildef

        twoseg  $psini
        reloc   $psini

        entry   $CMPOST,$CMPSTR

;       ROUTINE $CMPOST, ARG a-b, RESULT t, LOCAL a-e

;       strings are converted to upper case
;       a & b contain pointers to (string) objects
;       t returns 0 if the strings are identical,
;                -1 if b > a
;                 1 if a > b

$CMPOST:push    p,e
        seto    e,              ;non-zero e means case independent compare
        jrst    CMPST0

;       ROUTINE $CMPSTR, ARG a-b, RESULT t, LOCAL a-e

;       a & b contain pointers to (string) objects
;       t returns 0 if the strings are identical,
;                -1 if b > a
;                 1 if a > b

$CMPSTR:push    p,e
        setz    e,              ;zero e means case-dependent compare

CMPST0: push    p,a
        push    p,b
        push    p,c
        push    p,d

        getlng  t,a
        getlng  t2,b
        getpnt  a,a
        getpnt  b,b     ;note a & b are made into byte pointers

; set t to length of shorter string
;  t2 to $CMPSTR value if shorter string contained in longer
```

```
        camg t, t2
        jrst   CMPST1
         movei t2,1
         jrst  CMPSLP

CMPST1: caml   t,t2
        jrst   CMPST2
         move  t,t2
         movni t2,1
         skipa

; Strings are equal length

CMPST2: setz    t2,
       jumpe    t,CMPDON               ;jump if one string is zero length

CMPSLP: ildb   c,a
        ildb   d,b
;
; change to upper case if non-zero e

        jumpe  e,CMPSL0
        cail   c,"a"
        caile  c,"z"
        skipa
        trz    c,40

        cail   d,"a"
        caile  d,"z"
        skipa
        trz    d,40

CMPSL0: camg c,d
        jrst   CMPSL1
         movei t,1
         jrst  CMPDON

CMPSL1: camg d,c
       jrst     CMPSL2
         movni t,1
         jrst  CMPDON

CMPSL2: sojg    t,CMPSLP

       move     t,t2    ;come here if shorter is substring of longer
CMPDON: pop     p,d
       pop      p,c
       pop      p,b
```

```
        pop     p,a
        pop     p,e
        popj    p,

        end
```

3.2.4 LILCNV.MAC

```
       title LILCNV--LIL conversion routines (KL version)

        search lildef,monsym

        twoseg  $psini
        reloc   $psini

        subttl $$I2S,ARG b-c, USES a-e

;       converts integer in b to (newly allocated) string (backpointer in c)

        entry $$I2S

$$I2S:   move   e,[ ^o444400,,cvtbuf## ] ;36-bit byte pointer to temp. buffer
         jumpge b,I2S01
         movei  a,"-"            ; if negative, insert minus sign into
         movem  a,CVTBUF##       ; conversion buffer
         ibp    e
         movm   b,b              ; make positive

I2S01:   movei  d,15             ; maximum length of string
         setz   a,               ; clear high order part of integer
         extend a,[ CVTBDO  "0"
                         z       ];do conversion into CVTBUF
         jrst   GIVEUP##         ; if conversion failed
         movei  a,1-cvtbuf(e)    ;move # of digits in integer to a
         movei  b,strtyp
         .pushj p,$$ALLOC##
         getpnt e,(c)            ; pointer to newly allocated string
         move   d,a              ; a & d have length of string
         move   b,[ ^o444400,,cvtbuf## ]
         extend a,[ MOVSLJ
                         z ]     ; transfer buffer to string
         jrst   GIVEUP           ; if move failed
         popj   p,

        subttl $$S2I -convert string to integer
;
;       ARGS: a-e, in the format of an CVIDBT arg. block
;       takes non-skip return if string referneced by arg. block
;       is not numeric.
;       A numeric string is any number of spaces, followed by
;       an optional sign (+ or -),and one or more digits


        entry   $$S2I

$$S2I:  extend  a,[ CVIDBT CVTTAB ]
        popj    p,              ; conversion terminated on non-numeric char.
        tlnn    a,400000        ; if S bit was set, digits were found
        popj    p,              ; else take failure return

        tlne    a,200000        ; if N bit was set, conversion terminated
        aos     (p)             ; after finding digits
        popj    p,              ; (so N & S must be set for sucess return)

        subttl Translation table for CVTBDT instruction

        .NONNUM==100017         ; translation code for ending conversion
                                ; on non-numeric digit

        .DIGIT==400000          ;digit sets N & translates

CVTTAB:: repeat 20,< xwd .NONNUM, .NONNUM
                >
        xwd     000017,.NONNUM  ; left half is space,ignore if before number

        repeat 4,< xwd .NONNUM,.NONNUM
                >

        xwd     .NONNUM,200017  ; + sign, clear M, terminate if S .  1
        xwd     .NONNUM,300017  ; - sign, set M, terminate if S is 1
        repeat  1,< xwd .NONNUM,.NONNUM
                >

        %%t==0
        repeat  5,< xwd .DIGIT+%%t, .DIGIT +%%t+1
                %%t==%%t+2 >
        repeat  55,< xwd .NONNUM,.NONNUM
                >
        END
```

3.2.5 LILDAT.MAC

```
       title LILDAT--Data area for LIL run-time environment
;       split into write-enabled & read-only segments

        search LILDEF

        Twoseg  $psini
        reloc   0

        subttl APR service & LIL error storage

$aprjp:: z                  ;address of user-defined APR service routine

intwrd:: z                  ;flags & pc on level 1 interrupt
intwr2:: z                  ;flags & pc on level 2 interrupt

softrp:: z                  ; non-zero on entry to $RINT if interrupt is
                            ; software-generated (via SPRINGTRAP)
clktim:: z                  ; time (rel. to sys startup) of next timeout
clknum:: z                  ; number of clock interrupts since LOGO started

$$ercd:: z
$$erad:: z                  ;LIL error cells

        subttl Stack & Buffers

$stack:: block stksiz

cvtbuf:: block 15           ;buffer for integer to string conversion

ACTAB:: block p-lilr0+1 ;store ACS after $INIT

ERRACS:: block p                  ;save registers on a giveup

;brkset:: block 1_<bytsiz-5> ;buffer for Search & Verify breakset bits

        subttl storage allocator data

LASTR4::z                         ; last value of r4 (frame pointer) for $POLL
HPBOT:: z                         ; bottom of heap
```

```
HPTOP:: z                       ; top of heap
STFREE::z                       ; lowest free object in heap
BIGSIZ::z                       ; distance between top of heap
                                ;  and top of LOGO stack
FRENAM:: exp null               ; head of symbol table free list
NMBOT:: z                       ; bottom of name space

MARK1:: z                       ; storage for MOVS,CATS,APPS sources
MARK2:: z                       ;to be adjusted in compaction

COMRNT::z                       ;store compaction runtime

comment/                LIL STORAGE MAP


        777777  *****************
                *               *
                *               *
                *               *
                        .
                        .
                        .
                *    PROGRAM    *
                *     CODE      *
                *       &       *
                *     READ-     *
                *     ONLY      *
                *     DATA      *
                *               *
        $PSINI->*****************
                *               *
                *               *
                        .
                        .
                        .
                *    UNUSED     *
                *               *
bottom of STACK *****************
                *               *
                *     LOGO      *
                *     STACK     *
                *               *
                *               *
                *               *
            R4->*       --------*----!
                *               *    !
                *               *    !
top of stack -->*****************<---
```

```
        ^       *               *
   B    !       *               *
   I    !       *               *
   G    !       *   AVAILABLE   *
   S    !       *     HEAP      *
   I    !       *       &       *
   Z    !       *     STACK     *
        v       *     SPACE     *
        HPTOP   *****************
                *               *
                *               *
                *_______________*
                *\\\\\\\\\\\\\\\*
                *_______________*
                *               *
                *_______________*
                *\\\\\\\\\\\\\\\*
                *\\\\\\\\\\\\\\\*
                *_______________*
                *               *
                *               *
                *_______________*
        STFREE->*\\\\\\\\\\\\\\\*
                *_______________*
                *               *
        HPBOT-->*****************
                *               *
                *    SYMBOL     *
                *    TABLE      *
                *    ENTRIES    *
                *               *
        NMBOT-->*****************
                *               *
                *    OTHER      *
                *   ASSEMBLY    *
                *     TIME      *
                *   ALLOCATED   *
                *    STORAGE    *
        0 ----->*****************
/

        subttl APR tables (start of read-only segment)

        RELOC $psini

levtab:: exp intwrd     ;store flags & pc
```

```
          exp intwr2
          z
chntab::  xwd 1,$$CKINT##
          xwd 1,$$WKINT##
          xwd 1,$$STINT##
          block 3
          xwd 1,aprtrp## ;psi #6 is APR,priority 1,(LIL)service routine address is aprtrp
         block   ^d15
         xwd     1,$$PAGE##         ;debugging page trap handler
         block   ^d14               ;reserve space for rest of table

rictab::  exp $$CKRIC##             ;routines to read the interrupt ID code
          exp $$WKRIC##
          exp $$STRIC##

         subttl $SEARCH tables
comment ?
;wrdset:: block 1_bytsiz

chrset:: block 1_bytsiz

         %%t1==wrdset
         %%t2==chrset
         %%pc==.
         %%word==brkset
         repeat 1_<bytsiz-5>,<

                %%bit==1
                repeat 40,<
                             reloc %%t1
                             exp %%word

                             reloc %%t2
                             exp %%bit
                             %%t1==%%t1+1
                             %%t2==%%t2+1
                             %%bit==%%bit_1
                        >
                %%word==%%word+1
                 >

         RELOC %%pc
?
         end ;lildat
```

3.2.6 LILENV.MAC

```
       title LILENV-- TOPS-20 host interface for the LIL run-time Environment

        search LILDEF,MONSYM
        search  CONSTS          ; (for character definitions)

        twoseg  $psini
        reloc   $psini

        define address(addr)
        ;addr--list of address expressions
        ;print addresses in addr
        <
         irp addr,<
                        hrroi   1,[asciz/ say @ loc:/]
                        psout
                        movei   1,.priou
                        hrrzi   2,addr
                        move    3,[xwd 400000,10]
                                                ;;unsigned,radix 8
                        nout
                        halt .                  ;;halt on nout error
                        hrroi   1,[asciz/
/]
                        psout
                >
        > ;end of address

        subttl LILBEG--initializes LIL environment

LILBEG:
;
; turn off TTY echo, disable output translation

        movei   1,.PRIIN
        movei   2,TT%WAK+<0*TT%ECO>+<.TTATE>B29
        sfmod
        dmove   2,[ byte (2)2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2
                    byte (2)2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2 ]
        sfcoc
        ;turn on trapping

DDTLAB: movei   1,400000                ;=current fork
        move    2,[xwd levtab,chntab]   ;get address of psi tables
        sir                             ;give addresses to TENEX
        eir                                 ;enable trapping
        move    1,[xwd .TICBK,1]            ;assign BREAK to channel 1
        ati                                 ;(weak interrupt)
        move    1,[xwd .TICCZ,2]            ;assign CTRL-Z to channel 2
        ati                                 ;(strong interrupt)
;       move    1,[xwd 400000,400000]       ; set BREAK as a deferred int.
;       move    2,[xwd 400000,1000]
;       hllz    3,2
;       stiw
        setom   softrp                      ; init. soft trap flag
        movei   1,400000
        move    2,[ xwd 704000,020000]      ;turn on...
        aic                                 ;... channels 0,1,2,6,22
;
; for debugging, set high segment to read-only

        move    1,[400000,,400]
        movsi   2,100000
        hrrz    3,.JBHRL##
RDONLY: spacs
        subi    3,1000
        cail    3,400000
        aoja    1,RDONLY
        popj    p,

        subttl LOGO starting points

ENTVEC: jrst    WARMST
        jrst    HOTST
        jrst    COLDST

;
;cold start

COLDST: reset
        move    p,[iowd stksiz,$stack]      ;setup LIL stack
        pushj   p,LILBEG                    ;init. LIL environment
        pushj   p,HPSTRT##                  ;initialize heap

        .pushj  p,$INIT##                   ;start via stack

        pushj   p,$COMPACT##    ;squeeze store (hopefully read-only)

        hrroi   a,[asciz\LOGO can be saved and/or continued\]
        psout
        movei   a,400000
        move    b,[xwd 3,ENTVEC]
        sevec
        move    a,[xwd lilr0,ACTAB##]
        blt     a,ACTAB+p-lilr0 ;store ACS
        haltf
        jrst    HOTST

WARMST: reset
        move    a,[xwd ACTAB,lilr0]
        blt     a,p                     ;restore ACS

        pushj   p,LILBEG
HOTST:  pushj   p,$START##
        haltf
        jrst    .-2                     ; allows continuation after halt

        subttl APRTRP--arithmetic error trapping service routine

        generr(ovl,5001)                        ; arithmetic overflow code
        generr(div,5002)                        ; divide check code

aprtrp:: move   a,intwrd##
         movei  c,e.ovl         ; overflow code
         tlne a,40              ; test psw flags for divide check
         movei  c,e.div         ; c gets divide check error code
         skipe  b,$aprjp##      ; b is set to $aprjp (LIL trap address)
         jrst   TRPSET

; if trap has not been set

        hrroi   a,[asciz/arithmetic error /]
                                        ;now print adr of error

        address <@intwrd>              ;print address of APR

        haltf                           ; stop
        debrk                           ;return from interrupt if continued


TRPSET: movem   c,$$ERCD##                      ; push error code
        movem   a,$$ERAD##                      ; push addr. of error
        movem   b,intwrd                ; change trap break to $aprjp address

        debrk                           ;return from interrupt to
                                        ;user-specified location

        subttl LILIO-LIL host I/O routines

;       ROUTINE $RDCH, ARG lilr1, RESULT lilr0, LOCAL a-b
;       lilr1—jfn (fd)
;       reads character from jfn & returns it to lilr0

        entry $RDCH

$RDCH:  move    a,lilr1
        bin
        move    lilr0 ,b
;
;special handling of <CR><LF>

        caie    b,CR            ; if char. read was carriage return
        popj    p,              ; read in next char. & check if line feed
        bin
        cain    b,LF            ; if line feed,just return
        popj    p,
        bkjfn                   ;else back up to re-read
        halt    .
        popj    p,

;       ROUTINE $WRCH,ARG lilr1-lilr2, LOCAL a-b

;       lilr1—jfn (fd)
;       lilr2—character to be output

        entry $WRCH

$WRCH:  move    a,lilr1
        move    b,lilr2
        bout
        caie    b,CR                    ;append LF to CR's
        popj    p,
        movei   b,LF
        bout
        popj    p,

;       ROUTINE $OPEN, ARG lilr1-lilr2,RESULT lilr0,USES a-c

;       r1—pointer to filespec string
;       r2—mode word (one of IORD,IOWR,IOAPP,IOINT)
;
;       open file specified by r1 in mode specified in r2
;       (string must contain terminator character . See description of GETJFN
;       returns JFN in r0, or negative of error code if OPEN fails


        entry $OPEN
```

2

```
$OPEN:  getpnt  b,rl               ;get TOPS-20 byte pointer to filespec string

        caie    r2,iowr            ; open for writing?
        jrst    $OPEN1
         movsi  a,<gj%sht+gj%fou>_-22
         movei  c,of%wr
         jrst   OPENIT

$OPEN1: caie    r2,iord            ; open for reading?
        jrst    $OPEN2
         movsi  a,<gj%sht+gj%old>_-22
         movei  c,of%rd
         jrst   OPENIT

$OPEN2: caie    r2,ioapp           ; open for appending?
        jrst    $OPEN3
         movsi  a,<gj%sht>_-22
         movei  c,of%app
         jrst   OPENIT

$OPEN3: cain    r2,ioint           ; (interactive file .i.e. tty)
        jrst    .+3
        caie    r2,iornd           ; or random access
        jrst    OMODER

        movsi   a,<gj%sht> -22
        movei   c,of%rd+of%wr      ; open random & interactive files
         jrst   OPENIT             ; for both read & write access

OMODER:  movni  lilr0,1            ;unknown mode,return error code
         popj   p,


OPENIT: gtjfn
        erjmp [ movn     lilr0,a ;return negative of TOPS-20 error code
                popj     p,
              ]

        move    b,c                ;get OPENF flag word
        hrli    b,70000            ;set byte size to 7
        openf
        movn    a,a                ; on error, negate error code
        hrrz    lilr0,a            ;return jfn

        popj    p,                 ; return from $OPEN
```

```
;       ROUTINE $CLOSE,ARG r1-r2,LOCAL a

;       r1--File descriptor
;       r2--function code (IOCLS, normal close; IOABT, abort newly
;                          created output file; IODEL, delete file)

        entry $CLOSE

$CLOSE: hrrz    a,r1
        caie    r2,IODEL
        jrst    CLOSE1

        delf
        erjmp [ hrroi   a,[asciz / DELETE ERROR/]
                psout
                popj    p,
              ]

CLOSE1: cain    r2,IOABT
        hrli    a,4000
        closf
        ercal [ hrroi   a,[asciz/ CLOSE ERROR/]
                psout           ;print error message but return anyway on error
              ]
        popj    p,

;       ROUTINE $RENAME, ARG lilr1-lilr2, no result, uses a-c

;       r1--pointer to name of file to be renamed
;       r2--pointer to new name of source file
;
;       both files must be closed

        entry   $RENAME

$RENAME:
        getpnt  b,r1            ; get byte pointer to source string
        movsi   a,<gj%sht+gj%old>_-22
        gtjfn
        erjmp [ hrroi   a,[asciz /RENAME--BAD SOURCE FILE/]
                psout
                popj    p,
              ]
        move    c,a             ; save JFN of source
        getpnt  b,r2            ; now get JFN for dest.
        movsi   a,<gj%sht+gj%fou>_-22
        gtjfn
        erjmp [ hrroi   a,[asciz /RENAME--BAD DEST FILE/]
```

```
                psout
                popj    p,
              ]
        hrrz    b,a
        hrrz    a,c
        rnamf
        erjmp [ hrroi   a,[asciz /RENAMF ERROR/]
                psout
                popj    p,
             ]
        move    a,b                     ; release renamed file's JFN
        rljfn
        halt    .
        popj    p,
        subttl  $SENSE

;       r1 — FD (jfn) of file
;       r2 — item to be sensed
;       r0 — returned value of sensed item

        entry   $SENSE

$SENSE: setz    r0,                     ; default return value is 0
        move    a,r1                    ; get jfn
        caie    r2,IOPTR
        cain    r2,IOSIZ
        jrst    SNSPTR

        cain    r2,IOCOL
        jrst    SNSCOL

        cain    r2,IOWID
        jrst    SNSWID

        cain    r2,IOCHR
        jrst    SNSCHR

        gtsts                           ; status bits in b
        movsi   c,1000                  ; eof flag in status word
        caie    r2,IOEOF
        movsi   c,400                   ; error flag

        tdne    b,c                     ; test for flag
        movei   r0,1
        popj    p,

SNSPTR: rfptr                           ; get file position pointer
        popj    p,                      ; if error, just return default
        cain    r2,IOSIZ
        jrst    SNSSIZ
        move    r0,b
        popj    p,              ; just return pointer for IOPTR

SNSSIZ: move    c,b             ; store original position of ptr.
        movni   b,1
        sfptr                   ; move pointer to EOF
        popj    p,
        rfptr                   ; read EOF pointer
        popj    p,
        move    r0,b
        move    b,c             ; restore old pointer
        sfptr
        popj    p,
        popj    p,

;
;use mtopr as soon as release 4 is up
; till then, use rfmod

SNSWID: dvchr                   ;get characteristics of file's device
        hlrz    a,a             ;get device type code in rh of a
        jumpe   a,SNSWI2        ;if code=0, then controlling TTY
        caie    a,.DVDES+.DVTTY
        popj    p,              ;if not TTY, return 0 width

SNSWI2: move    a,r1            ;get JFN again

;       movei   b,30            ; (get line width)
;       mtopr
;       move    r0,c
;       popj    p,

        rfmod
        ldb     r0,[ point 7,b,17 ]
        popj    p,

SNSCOL: rfpos
        hrrz    r0,b            ; column in right half of b
        popj    p,

SNSCHR: sibe                    ; skip if no character available
        movei   r0,1
        popj    p,

        subttl $CONTROL
```

```
;       r1 — FD (jfn) of file
;       r2 — control item
;       r3 — new value of item

        entry   $CONTROL

$CONTROL:
        move    a,r1
        cain    r2,IOPTR
        jrst    CTLPTR

        cain    r2,IOSIz
        jrst    CTLSIZ

        cain    r2,IOERR
        jrst    CTLERR

        cain    r2,IOWID
        jrst    CTLWID

        popj    p,                      ; return if no legal item given

CTLPTR: move    b,r3
        sfptr
        popj    p,
        popj    p,

CTLSIZ: hrli    a,.fbsiz
        seto    b,
        move    c,r3
        chfdb
        popj    p,

CTLERR: gtsts
        tlz     b,400                   ; turn off error bit
        ststs
        popj    p,
        popj    p,
;
;use mtopr when release 4 is up

CTLWID: ;movei  b,31                    ; set page width
;       move    c,r3
;       mtopr
        rfmod
        dpb     r3,[ point 7,b,17 ]
        stpar
        popj    p,
        subttl .JSERR — JSYS ERROR HANDLER
;
; NOW HANDLES LIL ERRORS CORRECTLY (sets error cells, not stack)
;
;        LUUO to handle error returns from JSYS's,translating them into
;        LIL error codes & setting up the error cells. This can be used
;        anytime a single error instruction is needed.
;
;       The effective address of the LUUO is either the LIL error code
;       or the JSYS error code, depending on the op code ($JSERR or $LERR)
;       The address of the error jump is in the A register of the LUUO

        loc 41
        .pushj  p,.JSERR                ; LUUO handler

        reloc

.JSERR::
        setzb   a,b                     ; clear a & b
        ldb     a,[ point 9,40,8 ]      ; get LUUO op code
        ldb     b,[ point 4,40,12]      ; get A register (error jump addr.)

        caie    a,%JSERR
        jrst    LILERR

;       insert code to look up jsys & jsys error codes
;       then translate into LIL error code
;       for now just use JSYS code

        hrrz    c,40                    ; get code from right 1/2 of instr.
        pop     p,d                     ; get addr. of error
        jrst    ERRET

LILERR: hrrz    c,40                    ; get code
        pop     p,d                     ; get addr. of error

ERRET:  movem   c,$$ERCD                ; store error code
        movem   d,$$ERAD                ; store error addr.
        jrst    @(b)                   ; jump to error routine addr.

        subttl  LOGO asynchronous interrupts

        entry   $RINT

; $RINT — receive interrupt
;     Called from TRPHDL with the $PI code in rl. Disables the interrupt
;     channel on which the trap occurred, places a trap-specific code in
;     r0, and then (unless the trap was sprung via SPRINGTRAP) re-enables
;     the interrupt system by returning to TRPHDL with a DEBRK.
;       Can be called with any interrupt code, but a no-op for interrupt
;     codes <= T$ERR

$RINT:  hrrei   t,-<T$ERR+1>(r1) ;get bit position of trap channel
        skipge  t
        popj    p,               ;return if code too low
        movn    t2,t
        movei   a,400000         ; disable trapped channel
        movsi   b,400000         ;set up trap mask
        lsh     b,(t2)
        dic
        pushj   p,@rictab(t)    ; get ID code in r0
        move    t,softrp
        setom   softrp           ; was this a software-sprung trap???
        skipe   t
        popj    p,                      ; yes, return via POPJ...
        pop     p,intwrd                ; no, return via DEBRK...
        debrk

        entry   $DINT

; $DINT -- dismiss interrupt
;    called by DISMISSTRAP and CLEARTRAP to indicate that processing of
;    a trap by LOGO is complete.  The $PI code of the trap will be in r1
;    upon entry to $DINT, which re-enable the interrupt channel on which
;    the trap occurred.  Can be called with any trap code, but a no-op if
;    code <= T$ERR

$DINT:  movei   a,400000         ; fork handle  for this process
        movsi   b,400000         ; set up channel mask
        movni   t,-<T$ERR+1>(r1)
        lsh     b,(t)
        aic
        popj    p,

        entry   $POLL
        entry   $CLINT,$$CKINT,$$WKINT,$$STINT
        entry   $$CKRIC,$$WKRIC,$$STRIC
;
;On TOPS-20, $POLL checks for impending store overflows
; sets the $PI word with the store overflow code if so found
;$POLL is only useful if the stack grows without invoking the storage
; allocator (for example: TO FOO
;                         10 foo
;                         end )
;Therefore, to avoid duplicating too often the checking done in LILALC
```

```
; $POLL keeps track of where the stack frame pointer (R4) was the last time
; it was called, and only does serious work when it grows upward.
;
; calls $COMPACT in LILALC

$POLL:  camn    r4,LASTR4##
        popj    p,                  ;if no change in r4, return
        caml    r4,LASTR4
        jrst    POLLDN              ;jump if stack is shrinking
;
;stack is growing, check distance twixt heap & stack

        move    t,(r4)              ;get top of stack addr.
        sub     t,HPTOP##
        caile   t,DANGER            ;too close?
        jrst    POLLDN              ;no, return
        pushj   p,$COMPAC           ;yes, compact (& set BIGSIZ)
        move    t,BIGSIZ##          ;new distance twixt stack & heap
        caile   t,WARN
        jrst    POLLDN
        movei   t,T$ALC             ;still too close, set $PI
        movem   t,$PI##

POLLDN: movem   r4,LASTR4           ; make note of new r4
        popj    p,
;
;
$CLINT: pushj   p,$CCLOCK                   ; clear any pending timer int's
        cis                                 ;clear all interrupts
        popj    p,
;
; receive clock interrupt

$$CKINT:
        push    p,t
        setzm   softrp              ; a "real trap"!
        movei   t,T$TIME
        movem   t,$PI##
        aos     clknum              ; bump interrupt counter
        pop     p,t
        jrst    @intwrd

$$CKRIC:
        move    r0,clknum                   ; get # of times int has occurred
        popj    p,

;
; receive weak interrupt
```

```
$$WKINT:push    p,t
        setzm   softrp
        movei   t,T$KBD
        movem   t,$PI
        pop     p,t
        jrst    @intwrd

$$WKRIC:
        setz    r0,
        popj    p,

;
;receive strong (panic) interrupt

$$STINT:push    p,t
        setzm   softrp
        movei   t,T$PANC
        movem   t,$PI
        pop     p,t
        jrst    @intwrd

$$STRIC:
        movei   r0,32
        popj    p,

;
;debugging page fault

        entry   $$PAGE

$$PAGE: push    p,1
        push    p,2
        movei   1,400000
        gtrpw                           ;get trap status in AC1
        tlz     1,-1                    ;clear left 1/2
        cail    1,776000
        jrst    PAGERR                          ; fake pointer reference
        caig    1,STACK
        jrst    PAGDON
        cail    1,400000
        jrst    PAGERR
PAGDON: pop     p,2
        pop     p,1
        debrk
;
;reference to bad page
```

```
PAGERR: push    p,1
        hrroi   1,[asciz/Bad page reference.../]
        psout
        pop     p,1
        jrst    GIVEUP

        subttl  LOGO clock and timer routines

        entry   $CLOCK

; $CLOCK sets the timer to interrupt in (rl) milliseconds.

$CLOCK: pushj   p,$CCLOCK       ; clear any requests still waiting
        move    a,[xwd 400000,1]
        move    b,rl
        setz    c,
        timer
        halt    .
        time                            ; calculate time at which the
        add     a,rl                    ; interrupt will occur, so that
        movem   a,clktim                ; $RCLOCK can return a value
        popj    p,

        entry   $RCLOCK

; $RCLOCK -- returns the amount of time until the last call to $CLOCK
;       times out and interrupts.

$RCLOCK:
        move    r0,clktim               ; get time of interrupt
        time
        sub     r0,a                    ; and subtract present time
        skipge  r0                      ; if result is negative,
        setz    r0,                     ; return zero.
        popj    p,

        entry   $CCLOCK

; $CCLOCK -- clears any pending timer requests

$CCLOCK:
        move    a,[xwd 400000,5]
        timer
        halt    .
        setzm   clktim                  ; reset timer
        popj    p,

        entry   $elapsed
```

```
; $ELAPSED -- returns the time since the last system load as a pair
;       of integers; upon return, r0 has the # of minutes and r1 has
;       the # of seconds.

$ELAPSED:
        time
        move    r0,a
        idiv    r0,b
        idivi   r0,^d60
        popj    p,

        entry   $DELAY

; $DELAY -- dismisses the job for (r1) seconds

$DELAY: move    a,r1
        imuli   a,^d1000
        disms
        popj    p,

        entry   $TIME

; $TIME -- returns the time of day as a sequence of 4 integers; the
;       hour, minute, second, millisecond past midnight are returned in
;       r0 through r3.

$TIME:  seto    b,
        setz    d,
        odcnv
        hrrz    t,d
        setz    r3,
        idivi   t,^d60
        move    r2,t2
        idivi   t,^d60
        move    r1,t2
        move    r0,t
        popj    p,

        entry   $DATE

; $DATE -- returns the date as 3 integers: month, day, and year in r0-r2

$DATE:  seto    b,
        setz    d,
        odcnv
        hlrz    r0,b
        hrrz    r1,b
        aos     r1
        hlrz    r2,c
        aos     r2
        popj    p,

        subttl  LOGO is very sick

        entry giveup

GIVEUP: move    0,[xwd 1,ERRACS##]
        blt     0,ERRACS+p-1
        hrroi   a,[asciz/Internal error in LOGO.  Registers saved in ERRACS/]
        psout
        halt

        end COLDST
```

3.2.7 LILSYM.MAC

```
       title LILSYM--LIL symbol table manipulation routines

        search lildef,MONSYM

        opdef   adjbp [ ibp ]    ; TOPS-20 only

        twoseg  $psini
        reloc   $psini

;       ROUTINE $HASH, RESULT t, ARG a-b, LOCAL c-d, USES t2

;       a--address of name table
;       b--ptr to (string) data object
;       returns address in name table in t

        entry   $HASH

        define HSHCHR           ; special macro for $HASH, picks up 1 char.
        <
        rot     t,4
        ldb     t2,c
        trz     t2,40           ;; make upper case for HASHING purposes
        xor     t,t2
        >
```

```
$HASH:  getpnt  c,b                 ;c <- ptr to 1st byte in string
        getlng  d,b                 ; no. of bytes in string
        jumpn   d,HASH1
        move    t,a                 ; The empty string hashes to 1st
        popj    p,                  ; location in name table

HASH1:  move    e,d
        ash     e,-1                ; divide by 2
        caie    d,1
        subi    e,1                 ; e is amt. to adjust byte ptr.
                                    ; to approx. middle of string
                                    ; ( (length / 2) -1 ) )
        setzb   t,t2

        ibp     c                   ; c points to 1st byte
        HSHCHR                      ; get value of 1st char.

        push    p,e
        adjbp   e,c                 ; move byte ptr. to middle of string
        move    c,e
        pop     p,e

        HSHCHR

        caie    d,1
        ibp     c                   ; get next byte
        HSHCHR

        adjbp   e,c                 ; move byte ptr. to end of string
        move    c,e

        HSHCHR

        move    t2,(a)              ;get NINTS value
        idivi   t,-1(t2)            ;divide hash number by no. of hash buckets
        movm    t,t2                ;absolute value of remainder of division
        addi    t,1(a)              ;calc. pointer into table

        popj    p,

;       ENDROUTINE $HASH

;       ROUTINE $FIND, RESULT t, ARG a-b, LOCAL c-e, USES t2

;       a—address in name table(head of symbol chain)
;       b—ptr to (string) data object
;       returns "null" if string b is not in name table
;       otherwise returns pointer to symbol block which contains b

        entry   $FIND

$FIND:  push    p,c
        push    p,d
        push    p,e

        move    c,(a)           ;move addr. of 1st block into c
        movei   t,NULL
        cain    c,NULL          ;if no symbols in this bucket
         jrst   FINDON          ;return with NULL

        ;symbols with same hash code exist

        move    e,a             ;save a
        movei   d,null          ;d is ptr to last accessed symbol block

FIND1:
         hlrz   a,(c)           ;c points to current symbol block
         pushj  p,$CMPOST##     ;compare without regard to case
;
;  IF   t .EQ. 0 then string found in chain

        jumpn   t,FINNXT
;
; IF d .NE. #null then string not found at head of chain

        cain    d,NULL
        jrst    FIND2
                                ;so move block to head

            hrrz t,1(c)         ;get ptr to next block in chain
            hrrm t,1(d)         ;store in previous block

            move t,(e)          ;get addr. of former head of chain
            hrrm t,1(c)         ;link new head with former head

            movem c,(e)         ;insert new head in name table

FIND2:     move  t,c            ;return  addr. of found block

           move a,e             ;restore a
           jrst FINDON

;
;       string not found,move to next block
```

310

```
FINNXT:  move   d,c
         hrrz   c,1(c)

         caie   c,NULL
         jrst   FIND1

        movei   t,null          ;failure return
        move    a,e             ;restore a

FINDON: pop     p,e
        pop     p,d
        pop     p,c
        popj    p,


        subttl $ENTER, ARG a-c, USES t-t2

;       a--addr. in name table
;       b--ptr to (string) data object
;       c--addr. of ptr to (new) symbol block
;       if b is not found in symbol block chain @ a,enter b in (new)
;        symbol block at head of a (& give skip return)
;       otherwise, give an error return (no skip)
;       always return pointer to block containing ptr to string object
;       to (c)

        entry   $ENTER

$ENTER: pushj   p, $FIND
;
;IF     t <> NULL then string found, return addr. of found block to pointer

        cain    t,NULL
        jrst    ENTER1
         movem  t,(c)
         popj   p,

ENTER1:  pushj  p,$ALCBLK##     ;($ALCBLK takes arg. in c)
         move   t2,(c)          ;get pointer to new block
         move   t,t2

         exch   t,(a)           ;get old head of chain & place new head in table
         hrli   t,inflgs        ;set up flags in (2nd word of) symbol block
         movem  t,1(t2)

         movei  t,null          ;value ptr
         hrl    t,b             ;name ptr
         movem  t,(t2)
```

```
;
; do a SWAPP on name pointer

        exch    b,c             ; get string pointer in c
        pushj   p,HPJECT##
        jrst    ENTER2          ; don't set backpointer if not heapject


         hlrz   t,-1(c)         ; old owner of name
         hrlm   t2,-1(c)        ;make symbol block new owner
         movei  t2,NULL
         movem  t2,(t)          ; old owner now has NULL
ENTER2: aos     (p)             ;give skip return
        popj    p,

        subttl $REMOVE, USES t-t2, ARG a-b, LOCAL c

;       a—addr. in name table
;       b—ptr to (string) data object
;       removes block which contains b from name table
;       skip return if successful

        entry   $REMOVE

$REMOVE: pushj  p, $FIND        ;b is placed at head of chain by $FIND

;        IF t = #null   then $FIND was unsuccessful so return
        cain    t,NULL
        popj    p,

        hrrz    t2,1(t)         ;get pointer to next block
        movem   t2,(a)          ;make it head of table entry list
;
;now place removed block at head of free list
; (this was formerly $RETBLK in LILALC)

        move    t2,FRENAM##
        hrrzm   t2,1(t)         ;link field now points to old free list
                                ;flag field is zeroed
        movem   t,FRENAM        ;make block new head of free list
;
;deallocate name,value fields

        hrrz    a,(t)
        pushj   p,$$DEALC##
        hlrz    a,(t)
        pushj   p,$$DEALC
        aos     (p)             ;skip return for success
        popj    p,

        END
```

3.2.8 LILVEC.MAC

```
       title LILVEC — LIL vector subroutines

        search LILDEF

        twoseg $psini
        reloc  $psini


        entry $$RPLV            ; performs RPLVEC operation

; input regs:
;       a— addr. of destination vector
;       b— addr. of source buffer
;       c— position in vector of 1st word to be transferred
;       d— # of words to transfer


;       returns t as error flag (-1 for error)

; performs as much of the transfer as is possible.

$$RPLV: sojl    c,RPLERR        ; failure return if offset non-positive
                                ; also converts c to offset
        setz    t,              ; clear error flag
        getlng  t2,a            ; (t2)=vector length
        sub     t2,c            ; maximum legal value for d
        camg    d,t2
        jrst    RPLBLT
        seto    t,              ; too many words: flag error
        move    d,t2            ;  & adjust d

; transfer integer part of vector with BLT

RPLBLT: getfgs  e,a             ;get flags
        trne    e,numtyp        ;if number, transfer all with blt
        jrst    RPLNUM
        jumpg   c,RPLBL1        ; if NINTS is to be transferred,
        move    t2,(b)          ; move it first, so data in vector
        movem   t2,(a)          ; can be mapped in integers & pointers
        aoj     a,              ; incr. source & dest. addrs. to allow
        aoj     b,              ; for separate move
        aoj     c,              ; adjust starting offset
        soja    d,RPLBL2        ; decr. no. of words to move

RPLNUM: getlng  t2,a            ;for numbers, NINTS is length
        skipa
RPLBL1: move    t2,0(a)         ; get NINTS
        add     a,c             ; calc. starting addr. of dest.

RPLBL2: sub     t2,c            ; (t2)=max. # of ints. to transfer
        jumple  t2,RPLSWP       ; jump if no ints. to transfer
        camle   t2,d
        move    t2,d            ; entire transfer is integer
        sub     d,t2            ; decrement word count by transfer size
        hrl     e,b             ; source of BLT
        hrr     e,a             ; destination of BLT
        add     a,t2            ; incr. source,dest addresses by
        add     b,t2            ;  transfer size
        blt     e,-1(a)

RPLSWP: jumple  d,RPLDON
        movns   d               ; make count negative
        hrl     a,d             ; (a)= -count,,dest addr.
        move    e,b             ; source addr. in e

RPLLP:  push    p,t             ; save error flag

RPLLP2: hrrz    b,(e)           ; get source value
        pushj   p,$SWAPP##      ; do one-owner transfer (a,b are inputs)
        aoj     e,
        aobjn   a,RPLLP2        ; loop 'til count is zero

        pop     p,t             ; restore error flag

RPLDON: popj    p,              ; return

RPLERR: seto    t,              ; signal error
        popj    p,              ; return


        entry $$SUBV            ; performs RPLVEC operation when # of words
                                ; moved might be > 1
; input regs:
```

```
;       a— addr. of source vector
;       b— addr. of destination buffer
;       c— position in vector of 1st word to be transferred
;       d— # of words to transfer
;
;       returns t as error flag (-1 for error)
;
; performs as much of the transfer as is possible.
; can create transient pointers

$$SUBV: sojl    c,SUBERR        ; failure return if offset non-positive
                                ; also converts c to offset
        setz    t,              ; clear error flag
        getlng  t2,a            ; (t2)=vector length
        sub     t2,c            ; maximum legal value for d
        camg    d,t2
        jrst    SUBBLT
        seto    t,              ; too many words: flag error
        move    d,t2            ;  & adjust d

; transfer vector with BLT

SUBBLT: add     a,c             ; calc. starting addr. of source
        hrr     e,b             ; dest. of BLT
        hrl     e,a             ; source of BLT
        add     b,d             ; incr. dest. address by transfer size
        blt     e,-1(b)
SUBDON: popj    p,

SUBERR: seto    t,
        popj    p,

        end
```

4. LISTINGS OF THE LIL SOURCE FILES

4.1 INCLUDE modules

4.1.1 FLAGS.LIL

```
        search lil

        module  flags,include   ;Global flag definitions

; Bit definitions

        Const BIT$1,1
        Const BIT$2,BIT$1+BIT$1
        Const BIT$3,BIT$2+BIT$2
        Const BIT$4,BIT$3+BIT$3
        Const BIT$5,BIT$4+BIT$4
        Const BIT$6,BIT$5+BIT$5
        Const BIT$7,BIT$6+BIT$6
        Const BIT$8,BIT$7+BIT$7
        Const BIT$9,BIT$8+BIT$8
        Const BIT$10,BIT$9+BIT$9
        Const BIT$11,BIT$10+BIT$10
        Const BIT$12,BIT$11+BIT$11
        Const BIT$13,BIT$12+BIT$12
        Const BIT$14,BIT$13+BIT$13
        Const BIT$15,BIT$14+BIT$14
        Const BIT$16,BIT$15+BIT$15

;        Heapject prefix flags

        Const F$STR,BIT$1       ;heapject is a string
        Const F$VEC,BIT$2       ;heapject is a vector
        Const F$LIST,BIT$3      ;heapject (string) is a list
        Const F$NUM,BIT$4       ;heapject is number
        Const F$SIGN,BIT$5      ;number is negative

;       bits 6-8 are reserved in all heapjects

;        Procedure directory flags

        Const F$PRD, BIT$9      ;procedure has no read access
        Const F$PWRT,BIT$10     ;    "      "     write access
        Const F$PXCT,BIT$11     ;procedure has no execution access
        Const F$BURY,BIT$12     ;     "    does not have source code in
                                ;                    directory
        Const F$TEXT,F$PRD+F$BURY ; is text available?
        Const F$PINV,BIT$13     ;procedure is invisible
        Const F$STEP,BIT$14     ;proc. is stepped

        Const F$CHGD,F$TEXT     ;flags which disappear when procedure is
                                ; redefined
;
; General symbol table entry flags

        Const F$MARK,BIT$8      ;for name table garbage collector
        Const F$PRIV,BIT$15     ;restricts access by non-privleged
        Const F$TRCE,BIT$16     ;Trace on (same for name table entries)

;       Procedure table entry flags

; no. of args to procedure kept in bits 1-7

        Const F$ARGS,BIT$1+BIT$2+BIT$3+BIT$4+BIT$5+BIT$6+BIT$7

        Const F$JMP,BIT$9       ;Procedure is part of Evaluator
                                ; (not called with PCALL)
        Const F$GRDY,BIT$10     ;Procedure is greedy
        Const F$PRIM,BIT$11     ;Procedure is primitive
        Const F$QUOT,BIT$12     ;Procedure is quoting

        Const F$NARG,BIT$13     ;    "   demands numeric args.
        Const F$SARG,BIT$14     ;    "       "    string  "
                                ;(if neither, then procedure doesn't care)

        Const F$NTRP,F$GRDY+F$QUOT+F$PRIM ;bits that prevent a procedure
                                          ;from being a trap
        Const F$CHGT,F$ARGS+F$GRDY+F$QUOT ;flags which disappear when
                                ; procedure is redefined

;       name table entry flags

        Const F$NWRT,BIT$9      ;name can't be written
        const F$NINV,BIT$10     ;name is invisible
        Const F$NRD,BIT$12      ;name can't be read

;        Buffer Manager Flags

        const   B$STR,BIT$1     ;Object is a string
        const   B$LAST,BIT$2    ;Last (byte,word) of object is now in buffer
        const   B$DONE,BIT$3    ;Object is closed or all read in

;
;       Eval stack frame flags (word EVFLGS)

; Mask to see if this frame received trap

        Const   EV$PI,BIT$1+BIT$2+BIT$3+BIT$4+BIT$5+BIT$6+BIT$7+BIT$8

        const   EV$RUN, BIT$9   ;this frame is a RUN procedure
        const   EV$VAL, BIT$10  ;this frame is a VALUEOF procedure
        const   EV$MON, BIT$11  ;this frame is part of the monitor
        const   EV$TRT, BIT$12  ;this frame will receive a TRAPRETURN
        const   EV$PRV, BIT$13  ;this frame is running in priv. mode
        const   EV$SPT, BIT$14  ; this frame called SPRINGTRAP

        const   EV$PRN, EV$RUN + EV$PRV ;privileged RUN (#PRIVRUN)

;
; Flags for special literal table entries
; Flags identify the primitive(s) for which the literal is an input

        const   F$FILE, BIT$1   ;file status primitives
        const   F$OPEN, BIT$2   ;OPEN command
        const   F$TRAP, BIT$3   ;trap handling primitives
        const   F$PTYP, BIT$4   ;procedure type literals
        const   F$NPRP, BIT$5   ;literal is property of a name
        const   F$PPRP, BIT$6   ;literal is property of procedure

        endmod  FLAGS
```

4.1.2 CONST.LIL

```
       search  LIL
        module  CONSTS, include
        include FLAGS

; This module contains the definitions of all implementation-dependent
; constants which are not defined by the local LIL machine

        const   VERNUM,0        ; this is the LOGO version number
        const   WORDSZ,1        ;this is the amount by the addresses
                                ;of two consecutive data words differ
                                ;i.e., this is the amount added to
                                ;the operand of a INCP instruction
      const   MWRDSZ,-WORDSZ

;
;     Special (printable) characters

      const   BEL, 7
      const   CR, 13
      const   TAB, 9
      const   LF, 10
      const   NAK, 21           ; line delete
      const   SPACE, " "
      const   C$CMNT, "!"              ;comment char. & terminator
      const   C$QUOT, 34
      const   C$OPEN, "("
      const   C$CLOS, ")"
      const   C$COMM, 44
      const   C$DOTS, ":"
      const   C$LBRK, "["
      const   C$RBRK, "]"
      const   C$LBRC, "{"
      const   C$RBRC, "}"
      const   C$LANG, 60        ; ASCII only
      const   C$RANG, 62        ; ASCII only
      const   C$SEMI, ";"
      const   C$MINS, "-"
      const   C$EQU,  "="
      const   C$CHRQ, "        ; character quote
      const   DEL, 127          ; character delete

      const   EOL, CR           ; end-of-line marker
;
;     Untouchable characters
;
; PDP-10 LIL employs 9-bit bytes. Only the untouchables can have BIT 9 on.

      const   U$SPCS,BIT$9 + 0         ;separates items in a list
      const   U$LBRK,BIT$9 + 1         ;marks the beginning of a sub-list
      const   U$RBRK,BIT$9 + 2         ;marks the end of a sub-list

;
;     Heap Buffer sizes
;
; These are the sizes of the various heap buffers throughout LOGO
; passed as BCB's (buffer control blocks) to the buffer management
; routines in module BUFMGR
```

```
        const   EVBLEN, 200             ; evaluator buffer length
                                        ; (buffer for LOGO source code passed
                                        ;  to PARSER )

        const   SBUFSZ, 200             ; size of the input & output buffers
                                        ; used by the string processing
                                        ; primitives
        const   RBUFSZ, 200             ; size of output buffer for reader

        const   PSBFSZ, 200             ; size of output parse vector buffer

        const   SCBSIZ, 200             ;size of scanner output buffer

        const   NSTBSZ, 50              ;size of number-to-string buffer

;
; hash table sizes

        const   RWTSIZ,50
        const   NTSIZ, 100
        const   PTSIZ, 100
        const   FTSIZ, 10
        const   SPTSIZ,20
;
; temporary stack start

        const   STACK,40000

;
;       Trap handler constants

; $PI channel values

        const   T$PRSE,1        ;parse error channel
        const   T$EVAL,2        ;evaluation error (improper input) channel
        const   T$FILE,3        ;file error channel
        const   T$ALC, 4        ; "store nearly full" interrupt


        const   T$TRCE,5        ; trap trace (demons)
        const   T$TIME,6        ;clock interrupt
        const   T$KBD, 7        ;(normal) keyboard interrupt
        const   T$PANC,8        ; panic channel
        const   T$WARN,9        ;warning channel

        const   T$ERR, 5        ; Channel values less than or equal to
                                ;  this are error interrupts
        const   NTRAPS,9        ;size of trap tables
```

```
;
; define offsets for constant part of stack frame
;  Remember, the stack grows DOWNWARD

p0:     int 0                               ;flags for stack frame
p1:     ptr null                            ;PT entry of procedure
                                            ; (NULL means RUN-type procedure
p2:     ptr null                            ;offset in proc. dir. to current line
                                            ; (ptr. to parse vector in RUN-types)
p3:     int 0                               ;offset to current CM instr. in
                                            ; parse vector
p4:     ptr null                            ;ptr. to TOP entry of previous stack
                                            ; frame
p5:     ptr null                            ;ptr. to top of local stack
p6:
        const EVFLGS,P5-p0
        const PROC,P5-p1
        const LINE,P5-p2
        const PSOFF,P5-p3
        const PREV,P5-p4
        const TOP,p5-p5

        const FRMSIZ,p6-p0        ;offset between 1st & last word in constant
                                  ;part of frame
;
;       IO constants

; IO modes

        CONST IO$RD,0
        CONST IO$WR,1
        CONST IO$APP,2
        CONST IO$INT,3
        CONST IO$RND,4

; sense & control items

        CONST IO$PTR,1
        CONST IO$EOF,2
        CONST IO$SIZ,3
        CONST IO$ERR,4
        CONST IO$WID,5
        CONST IO$COL,6
        CONST IO$CHR,7

; $close function codes
```

```
        CONST IO$CLS,0
        CONST IO$DEL,1
        CONST IO$ABT,2

;
;  indices to the property tables in LGONAM (values of the property literals)

        const   P$PRM,  0               ;primitive
        const   P$INV,  P$PRM + WORDSZ  ;invisible
        const   P$TRC,  P$INV + WORDSZ  ;traced
        const   P$STP,  P$TRC + WORDSZ  ;stepped
        const   P$NRD,  P$STP + WORDSZ  ;no read
        const   P$NWR,  P$NRD + WORDSZ  ;no write (no change)

;
;               PDP-10 ONLY,NOT PART OF LIL

        radix   8
                                ; all addresses specified are in Octal

;       local BASE for arithmetic minus 1
;       greatest positive integer that will fit in 1 word

        const   BASE1,377777777777

;       FAKE POINTERS


; In the BBN LOGO implementation, addresses 776000-777777 are made inaccessible
; all fake pointers have values in this range

;*1     const   NULL,   777777  ;this value is actually defined in BBN LIL
        const   UNDEF,  777776

; NUMERIC FAKE POINTERS
;
; page 776 is reserved for numeric fake pointers

        const   NUMFAK, 776000
        const   NUMTOP, 776777

; The definitions of CM instructions.
; Note that the CM instructions are to be used for indices into a table(JMPTAB)
;  of pointers in the evaluator, and thus may not be consecutive integers on
;  byte addressable machines

        const   CMFAKE, 777000
        const   O$LITC, OMFAKE
        const   O$VALC, O$LITC + WORDSZ
        const   O$DREF, O$VALC + WORDSZ
        const   O$JUMP, O$DREF + WORDSZ
        const   O$JMPI, O$JUMP + WORDSZ
        const   O$JMPW, O$JMPI + WORDSZ
        const   O$JMPU, O$JMPW + WORDSZ
        const   O$DECJ, O$JMPU + WORDSZ
        const   O$NCAL, O$DECJ + WORDSZ
        const   O$EOL,  O$NCAL + WORDSZ
        const   O$EOS,  O$EOL + WORDSZ

        const   O$NOOP, O$EOS + WORDSZ

        const   OMLAST, O$NOOP

        endmod  CONST
```

4.1.3 ERRORS.LIL

```
        search  LIL
         module  errors, include
         include flags

;
; Definitions of the LIL errors — Implementation dependent

;1) String move errors

         const   E.SOE,1001      ;string offset error

;2)  Symbol table errors

         const   E.ANP, 2000     ;advance null pointer (in GETNXT)

;3)  Vector move errors

         const   E.VOE, 3001     ;vector offset error

;4) Numeric string errors

         const   E.NNS1, 4001    ;(1st arg. was) non-numeric string
        const   E.NNS2, 4002    ;(2nd arg. was) non-numeric string

;5) Arithmetic program checks (all data types including string)

        const   E.OVL,  5001    ;overflow (or underflow)
        const   E.DIV,  5002    ;divide check

;
; Definitions of the LOGO errors which can be caused.
;
; All error codes are an offset which corresponds to the trap
; channel activated on the occurence of the error plus an index

;       Error offsets

        const   E$PRSE, 0
        const   E$EVAL, 100
        const   E$FILE, 200
        const   E$ALC,  300
        const   E$TRCE, 400
;
; Parse errors.

        const   E$CLOS, E$PRSE + 1      ; ")" expected.
        const   E$RBRC, E$PRSE + 2      ; "}" expected.
        const   E$DOEX, E$PRSE + 3      ; "do" expected.
        const   E$BDRF, E$PRSE + 4      ; Bad reference, eg :+.
        const   E$FNEX, E$PRSE + 5      ; Fn name expected.
        const   E$CMEX, E$PRSE + 6      ;command expected
        const   E$EREL, E$PRSE + 7      ;extra right bracket on list
        const   E$MREL, E$PRSE + 8      ;missing right bracket on list
        const   E$TFA,  E$PRSE + 9      ;too few args.
        const   E$EXEX, E$PRSE + 10     ;expression expected
        const   E$JUNK, E$PRSE + 11     ; Junk at end of line.
        const   E$BFN,  E$PRSE + 12     ;Badly formed number
;
;       Non-file errors detected by the evaluator
; (either by a failed primitive or inside the evaluator module)

        const   E$WELF, E$EVAL + 0      ; word expected, list found
                                        ;  (e.g. THING [ gargle ] )
        const   E$UNDF, E$EVAL + 1      ; attempt to get the 'thing' of
                                        ; an undefined name
        const   E$NBIC, E$EVAL + 2      ; non-boolean input to a conditional
                                        ; (e.g. IF "foo THEN ... )
        const   E$NENF, E$EVAL + 3      ; number expected, not found
                                        ; (e.g. SUM "a 5 )
        const   E$OIC,  E$EVAL + 4      ; operation instead of a command
        const   E$CIO,  E$EVAL + 5      ; command instead of operation

        const   E$WNA,  E$EVAL + 6      ; wrong number of arguments

        const   E$LEWF, E$EVAL + 7      ; list expected, word found
                                        ; (e.g. FIRSTPUT "FOO "word )
        const   E$IMPF, E$EVAL + 8      ; improper format for a special

        const   E$DIV0, E$EVAL + 9      ; division by 0

        const   E$WSZO, E$EVAL + 10     ; wrong size object
                                        ;e.g, ASCII takes 1 char. word
        const   E$AOR,  E$EVAL + 11     ;arg. out of range
                                        ;(e.g., SQRT -1)
        const   E$UNTT, E$EVAL + 12     ;unknown trap type (literal)

        const   E$PCBT, E$EVAL + 13     ;procedure can't be trap
                                        ;(it's primitive or has args.)
        const   E$NALN, E$EVAL + 14     ;not a line number

        const   E$NIAP, E$EVAL + 15     ;not in a procedure

        const   E$LNNF, E$EVAL + 16     ;line number not found

        const   E$TNIP, E$EVAL + 17     ;trap not in progress

        const   E$WENF, E$EVAL + 18     ;word expected, number found

        const   E$NXOB, E$EVAL + 19     ;non-existent object

        const   E$OBPV, E$EVAL + 20     ;object protection violated

        const   E$NXSP, E$EVAL + 21     ;non-existent status or property

        const   E$TXDP, E$EVAL + 22     ;text didn't parse (in NOTEXT)

        const   E$TPRC, E$EVAL + 23     ; procedure is a trap handler

        const   E$APRC, E$EVAL + 24     ; procedure is active
                                        ; (above two errors for ERASE)
;
; IO errors

        const   E$UNAT, E$FILE + 0      ;unknown access type for OPEN

        const   E$FAOP, E$FILE + 1      ;attempt to open file already open
        const   E$FNOP, E$FILE + 2       ;attempt to access unopened file

        const   E$FINC, E$FILE + 3       ;attempt to access file in incorrect
                                         ;mode
        const   E$IOBL, E$FILE + 4       ; ill-formed list read

        const   E$OFAI, E$FILE + 5       ;open failure
;
; trace traps

        const   E$TRCR, E$TRCE + 0       ;traced on variable read
        const   E$TRCW, E$TRCE + 1       ;traced on variable write
        const   E$TRCP, E$TRCE + 2       ;traced on procedure call
        const   E$TRCS, E$TRCE + 3       ;traced on stepped procedure line
        const   E$TRCF, E$TRCE + 4       ;trace on procedure finish

        endmod  errors
```

4.1.4 TOKENS.LIL

```
       search  LIL

        module  TOKENS, include

        include FLAGS
;
; token flags.
; Token values are composed of flag bits + a code.

        const   T$PUNC,BIT$9             ;token is punctuation (can end a word)
        const   T$EXPR,BIT$10            ;token can start an expression (if this bit
                               ; is not on, token ends greedy arg. lists)
        const   T$BOPR, BIT$11           ;token is a binary operator
        const   T$TYPE,BIT$6+BIT$7+BIT$8; mask for operator type
;
; mask for index into operator proc. table
        const   T$OPCD,BIT$1+BIT$2+BIT$3+BIT$4+BIT$5

        const   T$TPSH,BIT$6    ;amt. to shift operator type
        const   T$OPSH,BIT$1    ;amt. to shift operator code


        const   T$UOPR, BIT$12           ;token is a unary operator
```

```
; Token types for communicating with scanner.
; Keywords.
        const   T$IF, 1
        const   T$DO, 2
        const   T$ELSE, 3
        const   T$WHIL, 4
        const   T$UNTL, 5
        const   T$REPT, 6
        const   T$ITER, 7
        const   T$FROM, 8
        const   T$NEXT, 9

; binary operator types

        const   T$ASGN, 1*T$TPSH
        const   T$REL,  2*T$TPSH
        const   T$ADD,  3*T$TPSH
        const   T$MUL,  4*T$TPSH

        const   T$BCMS, T$OPSH*0 + T$BOPR + T$ASGN
        const   T$LESS, T$OPSH*5 + T$BOPR + T$REL
        const   T$GRTR, T$OPSH*6 + T$BOPR + T$REL
        const   T$EQAL, T$OPSH*7 + T$BOPR + T$REL
        const   T$SUM,  T$OPSH*8 + T$BOPR + T$ADD
        const   T$DIFF, T$OPSH*1 + T$BOPR + T$ADD
        const   T$PROD, T$OPSH*2 + T$BOPR + T$MUL
        const   T$QINT, T$OPSH*4 + T$BOPR + T$MUL
        const   T$REM, T$OPSH* 3 + T$BOPR + T$MUL
;
;prefix chars.

        const   T$UMIN, 24 + T$UOPR + T$EXPR
        const   T$DOTS, 33 + T$EXPR
;
; Punctuation.
        const   T$OPEN, 30 + T$PUNC + T$EXPR
        const   T$LBRC, 31 + T$PUNC
        const   T$COMM, 32 + T$PUNC
        const   T$SEMI, 35 + T$PUNC
        const   T$CLOS, 36 + T$PUNC
        const   T$RBRC, 37 + T$PUNC
        const   T$EOL,  38 + T$PUNC
;
; Data.
        const   T$LITR, 50 + T$EXPR     ;literal(quoted word, number,list)
        const   T$WORD, 51 + T$EXPR     ;unquoted word
```

```
;
; Artifacts of the scanner (never returned as tokens).

        const   T$DIGIT,71
        const   T$LIST, 72
        const   T$CMNT, 73 + T$PUNC
        const   T$QUOT, 74 + T$EXPR
        const   T$BLNK, 80 + T$PUNC
        const   T$IDCH, 81
        const   T$LBRK, 82 + T$PUNC
        const   T$RBRK, 83 + T$PUNC
        const   T$CHRQ, 84 + T$PUNC
;
; The fatal error token—halts further scanning

        const   T$FERR, -1

        endmod  TOKENS
```

4.2 Executable Modules

4.2.1 ARITH.LIL

```
        search  LIL

        module  Arith  — LOGO arithmetic utility routines

        include CONSTS
        include FLAGS

        extrn   GIVEUP                  ;(for debugging)

        Const   F$NEG,F$SIGN+F$NUM      ;flags to set in a negative number

;[
;routine:CRNUM —create a number, initially 0
;input reg: r1—length of number
;output reg:0
;attr:UNSAFE
;type:global utilty
;]
```

```
        vsect

TEMP:   ptr     NULL

        entry   CRNUM

        psect

CRNUM:  pushm   r2,r3
        movi    <#,F$NUM>,r2
        movp    <#,TEMP>,r3
        host    $ALLOC
        cmovp   TEMP,r0
        movi    r1,TEMP
        rplvec  <#,TEMP>,r0                 ;set number length
        popm    r2,r3
        return

;[
;routine:MULDIG --multplies number by (positive) constant plus offset
;               starting at specified digit
;input regs: r0 --number ptr, r1--constant, r2--offset,r3-starting digit
;output reg: r0 --new number (original lost)
;attr:UNSAFE
;type: local utility
;]

        vsect

NEWDIG: int     0
NUMLEN: int     0

        psect

MULDIG: subvec  r0,<#,NUMLEN>               ;get length of number

MULDI1: jmpgi   r3,NUMLEN,MULDON
        subvec  r0,<#,NEWDIG>,r3            ;get next digit
        udivi   r1,NEWDIG,r2                ;NEWDIG<-NEWDIG*r1+r2 (carry to r2)
        rplvec  <#,NEWDIG>,r0,r3            ;replace in number
        inci    r3
        jmp     MULDI1

;
; if carry exists, then number must be extended by 1 digit

MULDON: jmpzi   r2,MULEND                   ;jump if no carry
        movi    r2,NEWDIG
        inci    NUMLEN
        movi    NUMLEN,r1
        movi    <#,F$NUM>,r2
        movp    <#,TEMP>,r3
        movp    <#,NULL>,TEMP
        host    $ALLOC                     ;allocate new number
        insn    r0,TEMP                    ;insert old number
        gettyp  r0,r1                      ;get sign of old number
        swapp   TEMP,r0
        settyp  r1,r0                      ;give sign to new number
        rplvec  <#,NUMLEN>,r0              ;give new length
        rplvec  <#,NEWDIG>,r0,NUMLEN       ;add new digit at end

MULEND: return

;[
;ROUTINE: CVTCN —convert characters to number (currently integer only)
;input REGS: r1—address of routine providing char. stream
;output REG: r0—output number
;ATTR:UNSAFE
;TYPE:global utility
;CALLS:MULDIG
;]
        vsect

DIGCHR: ptr     NULL                       ;char. IO routine addr. goes here
NEWNUM: ptr     NULL                       ;points to no. being built

        psect

        entry   CVTCN

CVTCN:  pushm   r2,r3
        movp    r1,DIGCHR
        movi    <#,1>,r1                   ;create initial 0 length number
        call    CRNUM
        omovp   r0,NEWNUM

        movp    DIGCHR,r1
        call    <@,1>,CSNERR              ;get 1st char.,fail after last char.
        jmpei   <#,"+">,r0,FIRCHR          ;check for leading +
        jmpnei  <#,"-">,r0,CHKCHR          ;if not leading sign, read digits
        settyp  <#,F$NEG>,NEWNUM           ;- sign,set as negative
;
;get first char. of number with sign, must exist

FIRCHR: movp    DIGCHR,r1
        call    <@,1>,CSNERR
```

```
;
;main loop, digtize char. & add to number

CHKCHR: cvtdi   r0,CSNERR                       ;check if char. is digit
;
;multply number by 10 & and new digit char.

        movi    r0,r2
        omovp   NEWNUM,r0
        movi    <#,10>,r1
        movi    <#,2>,r3
        call    MULDIG
        omovp   r0,NEWNUM
;
;get next char.

        movp    DIGCHR,r1
        call    <@,1>,CSNDON                   ;done if no more chars.
        jmp     CHKCHR
;
;successful exit

CSNDON: omovp   NEWNUM,r0
        popm    r2,r3
        succeed
;
;ill-formed number string, fail

CSNERR: swapp   <#,NULL>,NEWNUM                 ;deallocate partial result
        popm    r2,r3
        fail

;[
;ROUTINE:CVTNC--convert number to characters
;input REGS: r0--number, r1--output character routine(takes char. in r2)
;output REG: r0--(contents destroyed),chars. output in reverse order
;ATTR:UNSAFE
;TYPE:global utility
;]
        entry   CVTNC

CVTNC:  pushm   r2,r3
        gettyp  r0,r3                           ;get sign
        pushi   r3                              ;save sign
        subvec  r0,<#,NUMLEN>                   ;get length of dividend
        omovp   r0,TEMP
;
;repeatedly divide number by 10, outputting remainder
```

```
CVINC1: movi    NUMLEN,r3
        movi    <#,0>,r2                ;initial remainder

CVINC2: jmpli   r3,<#,2>,CVINC3
        subvec  temp,<#,NEWDIG>,r3      ;get next digit
        ldivi   <#,10>,r2,NEWDIG        ;r2<-remainder,NEWDIG<-quotient
        rplvec  <#,NEWDIG>,temp,r3
        deci    r3
        jmp     CVINC2
;
;if highest order word of quotient is 0, decrement length of number

CVINC3: subvec  temp,<#,NEWDIG>,NUMLEN
        jmpnzi  NEWDIG,CVINC4
        deci    NUMLEN

CVINC4: cvtid   r2                      ;remainder is output digit
        call    <@,1>                  ;output digit
        jmpgi   NUMLEN,<#,1>,CVINC1     ;keep going if quotient > 0

        popi    r3                      ;retrieve sign
        tstb    <#,F$SIGN>,r3,CVINC5
        jmp     CVINC6
CVINC5: movi    <#,"-">,r2
        call    <@,1>                  ;output sign

CVINC6: swapp   <#,NULL>,temp           ;dealc. remnant of number
        popm    r2,r3
        return
;
;       String conversion routines

        vsect
;
;output buffer for number-to-string conversion

NSBSIZ: int     NSTBSZ
NSBCNT: int     0
NSBPTR: ptr     NULL
NSBOBJ: ptr     NULL
NSBOFF: ptr     NULL
NSBLNG: int     0
NSBFLG: int     0
NSBBUF: blki    NSTBSZ

        psect
;
```

```
; reverse buffer management routines used for output

        extrn   RBOINI,RBOUP,RPTBUF

;[
;ROUTINE: NSTOUT--output digit bytes for CVTNS
;input REG: r2--output byte
;ATTR: UNSAFE
;CALLS: RBOUP
;TYPE:  called by CVTNS
;]

NSTOUT: push    r1
        movp    <#,NSBBUF>,r1
        call    RBOUP
        pop     r1
        return

;[
;ROUTINE: CVTNS --convert number to string
;input REG: r1--number (contents destroyed)
;output REG: r0
;ATTR: Unsafe
;CALLS: CVTNC,RBOINI,RPTBUF
;]

        vsect

NUMTMP: ptr     NULL                    ;store input number temporarily

        psect

        entry   CVTNS
        extrn   EMPTY
;
; setup output buffer

CVTNS:  omovp   r1,NUMTMP               ;save arg
        movp    <#,NSBBUF>,r1
        movp    <#,EMPTY>,NSBOBJ
        call    RBOINI
        movi    <#,B$STR>,NSBFLG
        movi    <#,0>,NSBLNG

        omovp   NUMTMP,r0               ;get arg.
        movp    <#,NSTOUT>,r1
        call    CVTNC                   ;do actual conversion
        jmpzi   NSBCNT,CVINS1           ;output remaining chars.
        movp    <#,NSBBUF>,r1           ; in buffer
        call    RPTBUF

CVINS1: swapp   NSBOBJ,r0
        return

;[
;ROUTINE: STNIN--supplies chars. for string to number conversion
;output REG: character, fails if no more characters
;ATTR:SAFE
;TYPE: called by CVTSN
;GLOBALS: NSBOBJ,NSBOFF (setup by CVTSN)
;]

        vsect

STNCHR: int     0

        psect

STNIN:  inci    NSBOFF
        substr  NSBOBJ,<#,STNCHR>,NSBOFF,<#,1>,STNDON
        movi    STNCHR,r0
        succeed

STNDON: fail

;[
;ROUTINE: CVTSN --convert string to number (fail if not possible)
;input REG: r1--string
;output REG:r0--number,r1--input string
;ATTR:UNSAFE
;CALLS: CVTCN
;GLOBALS: NSBOBJ,NSBOFF
;]
        entry   CVTSN

CVTSN:  gettyp  r1,r0
        tstb    <#,F$LIST>,r0,CVSER2    ;can't convert list
        omovp   r1,NSBOBJ               ;store during conversion
        movi    <#,0>,NSBOFF            ;current offset to input string
        movp    <#,STNIN>,r1
        call    CVTCN,CVTSER            ;jump if conversion fails
        omovp   NSBOBJ,r1
        succeed

CVTSER: omovp   NSBOBJ,r1
```

```
CVSER2: fail

;
;[
;ROUTINE:DECMAG--decrement magnitude of number, fails if number is 0
;input REG: r0--number
;output reg:r0--possibly new number, else input number with least sig.
;               digit decremented
;ATTR:UNSAFE
;GLOBALS: NEWDIG,NUMLEN
;TYPE:global utility
;]
        entry   DECMAG

DECMAG: push    r1
        subvec  r0,<$,NUMLEN>   ;get length of number
        movi    <$,2>,r1
        jmpli   NUMLEN,r1,DECMER        ;jump if given 0

DECMLP: subvec  r0,<$,NEWDIG>,r1
        deci    NEWDIG
        jmpnmi  NEWDIG,DECMDN
        movi    <$,BASE1>,NEWDIG
        rplvec  <$,NEWDIG>,r0,r1
        inci    r1
        jmp     DECMLP

DECMDN: jmppi   NEWDIG,DECMD1
        jmpli   r1,NUMLEN,DECMD1
;
;most sig. digit in number was reduced to zero so shorten number

        deci    NUMLEN
        gettyp  r0,r1                   ;get sign
        omovp   r0,TEMP
        extn    TEMP,r0,<$,1>,NUMLEN
        rplvec  <$,NUMLEN>,r0
        swapp   <$,NULL>,TEMP
        jmpli   NUMLEN,<$,2>,DECMD2
        settyp  r1,r0                   ;set sign if result not 0
        jmp     DECMD2
;
;replace changed digit for the normal case

DECMD1: rplvec  <$,NEWDIG>,r0,r1

DECMD2: pop     r1
        succeed
```

```
DECMER: pop     r1
        fail

;
;       routines to do addition & subtraction
;       all input numbers are referenced via INDIRECT pointers

        vsect

DIG1:   int 0
DIG2:   int 0
LENG1:  int 0
LENG2:  int 0
CARRY:  int 0                           ;both carry & borrow
SIGDIG: int 0

        psect

;[
;ROUTINE:CMPMAG --compares the magnitude of two numbers
;input REGS: r1--1st number, r2--2nd number
;output REG: r0--0 if equal, positive if 1st > 2nd, negative if 2nd > 1st
;ATTR:SAFE
;TYPE: global utility
;]
        entry CMPMAG

CMPMAG: push    r3
        subvec  <@,1>,<#,DIG1>          ;get length of numbers
        subvec  <@,2>,<#,DIG2>
;
;compare magnitudes of individual digits
;1st time thru loop check lengths, subsequently check digits

        movi    DIG2,r3                 ;# of digits

CMPLP:  movi    DIG1,r0
        subi    DIG2,r0
        jmpnzi  r0,CMPDON               ;exit if no.s are different
        jmpngi  r3,<#,1>,CMPDON         ;exit if no more digits
        subvec  <@,1>,<#,DIG1>,r3
        subvec  <@,2>,<#,DIG2>,r3
        deci    r3
        jmp     CMPLP

CMPDON: pop     r3
        return
```

```
;[
;ROUTINE: ADDMAG — adds the magnitude of two numbers
;input REGS: r1 — 1st number, r2— 2nd number
;               magnitude of <@,1> is greater than magnitude of <@,2>
;output REG: r0 — unsigned sum
;ATTR: UNSAFE
;TYPE: local utility
;]
        vsect

TMPRST: ptr     NULL

        psect

ADDMAG: push    r3
        subvec  <@,1>,<#,LENG1>
        pushp   r1
        movi    LENG1,r1
        call    CRNUM
        popp    r1
        subvec  <@,2>,<#,LENG2>
        movi    <#,2>,r3                ;r3 as offset to next digit
;
; main loop—copy sum of digits from args as appropriate

        movi    <#,0>,CARRY
ADDMA1: jmpgi   r3,LENG1,ADDDON         ;jump if no more digits in larger #
        subvec  <@,1>,<#,DIG1>,r3
        laddi   CARRY,DIG1,ADCRY1
        movi    <#,0>,CARRY
        jmp     ADDMA2

ADCRY1: movi    <#,1>,CARRY             ;signal carry

ADDMA2: jmpgi   r3,LENG2,ADDMA3         ;jump if no more digits in smaller #
        subvec  <@,2>,<#,DIG2>,r3
        laddi   DIG2,DIG1,ADCRY2
        jmp     ADDMA3

ADCRY2: movi    <#,1>,CARRY             ;one time thru loop should never hit
                                        ; both ADCRY1 & ADCRY2
;output new digit to result

ADDMA3: rplvec  <#,DIG1>,r0,r3
        inci    r3
        jmp     ADDMA1
;
```

```
;if carry, extend result by one digit

ADDDON: jmpzi   CARRY,ADDDN2            ;jump if no carry
        inci    LENG1
        omovp   r0,TMPRST
        movi    LENG1,r1
        call    CRNUM                   ;get # with extra digit
        insn    TMPRST,r0               ;store most of result
        rplvec  <#,LENG1>,r0            ;store length
        rplvec  <#,CARRY>,r0,LENG1      ;store most sig. digit
        swapp   <#,NULL>,TMPRST

ADDDN2: pop     r3
        return

;[
;ROUTINE: SUBMAG --subtracts the magnitude of 2 numbers
;input REGS: r1--1st number, which is greater than, r2-2nd number
;output REGS:r0--unsigned difference, r1,r2--inputs
;ATTR: UNSAFE
;TYPE: local utility
;]

SUBMAG: push    r3
        subvec  <@,1>,<#,LENG1>
        pushp   r1
        movi    LENG1,r1
        call    CRNUM                   ;create result object with maximum
        popp    r1                      ; possible length
        subvec  <@,2>,<#,LENG2>
        movi    <#,1>,r3                ;r3 is index to next digit
        movi    <#,0>,CARRY             ;(actually borrow)
        movi    <#,1>,SIGDIG            ; position of highest
                                        ; non-zero result digit
;main loop

SUBMA1: inci    r3
        jmpgi   r3,LENG1,SUBDON
        subvec  <@,1>,<#,DIG1>,r3      ;get subtrahend(?) digit
        jmpgi   r3,LENG2,SUBMA2         ;jump if no more subtractor digits
        subvec  <@,2>,<#,DIG2>,r3      ;get subtractor digit
        laddi   CARRY,DIG2,SUBCRY       ;add borrow to digit
        jmp     SUBMA3                  ; overflow means borrow,low order
                                        ; subtrahend is 0
;
; no subtractor digits, just subtract borrow

SUBMA2: movi    CARRY,DIG2
```

```
SUBMA3: movi    <#,0>,CARRY
        subi    DIG2,DIG1
        jmpnmi  DIG1,SUBMA4             ;jump if no borrow
;
;form complement in local base

BORROW: addi    <#,BASE1>,DIG1          ;add Base-1 (largest positive no.)
        inci    DIG1

SUBCRY: movi    <#,1>,CARRY             ;set borrow

SUBMA4: rplvec  <#,DIG1>,r0,r3
        jmpzi   DIG1,SUBMA1             ;adjust SIGDIG if non-zero digit stored
        movi    r3,SIGDIG
        jmp     SUBMA1
;
;return significant digits of result

SUBDON: jmpei   LENG1,SIGDIG,SUBDN2     ;jump if all digits are significant
        cmovp   r0,TEMP
        extn    TEMP,r0,<#,1>,SIGDIG    ;extract sig. digits
        rplvec  <#,SIGDIG>,r0           ;set integer length word
        swapp   <#,NULL>,TEMP           ;dealc. unsquozed result

SUBDN2: pop     r3
        return

;[
;ROUTINE:ADDNUM—adds two signed numbers
;input REGS:r1,r2—numbers
;output REG:r0—signed sum
;ATTR:UNSAFE
;TYPE: global utility
;CALLS:CMPMAG,ADDMAG,SUBMAG
;]
        entry   ADDNUM

ADDNUM: push    r3                      ;r3 will have sign of result
        gettyp  <@,1>,r0
        andi    <#,F$SIGN>,r0
        gettyp  <@,2>,r3
        andi    <#,F$SIGN>,r3           ;r0,r3 have signs of inputs
        jmpnei  r0,r3,SUBNUM            ;numbers of opposite signs-subtract
        call    CMPMAG
        jmpnmi  r0,ADD1                 ;jump if 1st is greater than 2nd
        movp    r1,r0                   ;switch args. to ADDMAG
        movp    r2,r1
        movp    r0,r2
ADD1:   call    ADDMAG
        jmp     ADDEND
;
;subtract magnitudes

SUBNUM: call    CMPMAG
        jmpnpi  r0,SUB1
;
;1st num is greater than second, sign is complement of r3

        xori    <$,F$SIGN>,r3
        jmp     SUB2
;
;2nd num is greater than 1st, switch args.

SUB1:   movp    r1,r0
        movp    r2,r1
        movp    r0,r2

SUB2:   call    SUBMAG
;
;arithmetic done,set sign & return

ADDEND: gettyp  r0,r1
        setb    r3,r1
        settyp  r1,r0
        pop     r3
        return

;
;  Multiplication & Division
        vsect

INDEX2: int     0                       ;index to digits of <@,2>
PPREG:  ptr     NULL                    ;partial prod. register
PDIG:   int 0                           ;partial prod. digit
PINDEX: int     0                       ;index to partial prod.
RINDEX: int     0                       ;index to output number digits
OUTSGN: int     0                       ;sign bit of output

        psect

;[
;ROUTINE:MULNUM —multiplies two signed numbers
;input REGISTERS: r1,r2—indirect pointers  to numbers
;output REGISTER: r0—product
;ATTR:UNSAFE
;TYPE:global utility
;]
        entry   MULNUM

MULNUM: subvec  <@,1>,<$,LENG1>
        jmpei   LENG1,<$,1>,MULZER         ;return zero
        subvec  <@,2>,<$,LENG2>
        jmpei   LENG2,<$,1>,MULZER         ;return zero
        push    r3
        gettyp  <@,1>,OUTSGN
        gettyp  <@,2>,r0
        andi    <$,F$SIGN>,r0
        xori    r0,OUTSGN
;
;create temporary number of length of first operand
; create output number of minimum possible length

        pushm   r1,r2
        movi    LENG1,r1
        movi    LENG2,r2
        subi    <$,2>,r2
        jmpzi   r2,MULNU1                  ;if 2nd operand is 1 digit, partial
                                           ; product never needed
        call    CRNUM
        omovp   r0,PPREG
        addi    r2,r1                      ;make room for product with length
                                           ;(sum of the lengths of the inputs)-2
MULNU1: call    CRNUM
        popm    r1,r2

        movi    <$,2>,INDEX2
        movi    <$,1>,RINDEX
;
;outer loop--multiply all digits of <@,1> by <@,2>
;       all except first result digit go into partial product reg.
;       (except on last <@,2> digit)

MLOOP1: subvec  <@,2>,<$,DIG2>,INDEX2
        movi    <$,0>,CARRY
        movi    <$,1>,r3                   ;r3 is index to digits of <@,1>
;
;inner loop

MLOOP2: inci    r3
        jmpgi   r3,LENG1,ML2DON
        subvec  <@,1>,<$,DIG1>,r3
;
;if not first row, (long) add partial product to carry
```

```
        jmpei   INDEX2,<#,2>,ML20        ;no partial prod. 1st time thru
        subvec  PPREG,<#,PDIG>,r3        ;get part. prod. digit
        laddi   PDIG,CARRY,MLCRY         ;part. prod. & carry will be added
                                         ; to product of digits to form
                                         ; double word carry

ML20:   movi    <#,0>,PDIG               ;PDIG becomes high order bit of carry
        jmp     ML21

MLCRY:  movi    <#,1>,PDIG

ML21:   udivi   DIG2,DIG1,CARRY          ;actual multiply
        addi    PDIG,CARRY               ;add high order of old carry to new
                                         ; carry (can't overflow)
;decide fate of result digit (DIG1)

        jmpei   r3,<#,2>,MOUTDG
        jmpei   INDEX2,LENG2,MOUTDG
;
;store in partial prod.

        movi    r3,PINDEX
        deci    PINDEX
        rplvec  <#,DIG1>,PPREG,PINDEX
        jmp     MLOOP2
;
;output to result

MOUTDG: inci    RINDEX
        rplvec  <#,DIG1>,r0,RINDEX
        jmp     MLOOP2
;
;end of inner loop--normally carry goes to partial prod.
;       if end of whole thing, output carry if non-zero,
;       then return result

ML2DON: jmpei   INDEX2,LENG2,ML1DON      ;done with whole multiply
        inci    INDEX2
        movi    r3,PINDEX
        deci    PINDEX
        rplvec  <#,CARRY>,PPREG,PINDEX
        jmp     MLOOP1

ML1DON: jmpzi   CARRY,MLDON2
        swapp   r0,PPREG                 ;use PPREG as temporary result
        inci    RINDEX
        movi    RINDEX,r1
        call    CRNUM
        insn    PPREG,r0,<#,1>,GIVEUP   ;extract number output so far
        rplvec  <#,RINDEX>,r0           ;change length field
        rplvec  <#,CARRY>,r0,RINDEX     ;add most sig. digit

MLDON2: settyp  OUTSGN,r0
        swapp   <#,NULL>,PPREG
        pop     r3
        return
;
;return zero if either input is zero

MULZER: movi    <#,1>,r1
        call    CRNUM
        return

;[
;ROUTINE:DIVNUM --returns signed quotient, remainder of two numbers
;input REGISTERS:r1--dividend, r2--divisor (indirect pointers)
;output REGISTERS:r0--quotient, r1--remainder
;               fails on divide by 0
;ATTR:UNSAFE
;TYPE:global utility
;CALLS:CRNUM,MULNUM,ADDNUM
;]
        vsect

DIVDND: ptr     NULL                    ;current dividend ptr
DIVSOR:ptr      NULL                    ;indirect divisor ptr.
DLENG1: int     0                       ;length of current dividend
DLENG2: int     0                       ;length of divisor
DIVSGN: int     0                       ;sign of divisor
QUOT:   ptr     NULL                    ;temp. quotient
TNUM:   ptr     NULL                    ;temp. number
DDIG:   int     0                       ;most sig. digit of divisor

        psect

        entry   DIVNUM

DIVNUM: subvec  <@,2>,<#,DLENG2>       ;get length of divisor
        jmpli   DLENG2,<#,2>,DIVZER     ;divide check
        push    r3
        subvec  <@,2>,<#,DDIG>,DLENG2  ;most sig. digit
        gettyp  <@,2>,DIVSGN
        subvec  <@,1>,<#,DLENG1>       ;length of initial dividend
        extn    <@,1>,DIVDND,<#,1>,DLENG1 ;copy dividend
        gettyp  <@,1>,r0
        settyp  r0,DIVDND
        movp    r2,DIVSOR               ;save divisor
        movi    <#,1>,r1                ;create initial quotient of zero
        call    CRNUM
        omovp   r0,QUOT
;
;main loop—repeatedly divide current dividend by DDIG & add result to
;       quotient.  Subtract product of result & divisor from dividend
;       until dividend = 0.

;
;create partial quotient of current dividend and DDIG

DVLOOP: movi    DLENG1,r1
        subi    DLENG2,r1               ;get minimum length of partial quotient
        jmpmi   r1,DVDONE               ;jump if divisor greater than
        inci    r1                      ; dividend
        movi    <#,0>,r2                ;r2 will hold carry
        movi    DLENG1,r3               ;index to dividend
        subvec  DIVDND,<#,PDIG>,r3      ;get high order digit of dividend
        ldivi   DDIG,r2,PDIG
        jmpzi   PDIG,PQ1                ;if quotient of high order digits
        inci    r1                      ; is non-zero, increase length of
        jmp     PQ2                     ; partial quotient

PQ1:    jmpli   r1,<#,2>,DVDONE         ;finish if par. quot. is zero
        deci    r3                      ;else get quotient of 1st two
        subvec  DIVDND,<#,PDIG>,r3      ; high-order digits
        ldivi   DDIG,r2,PDIG

PQ2:    call    CRNUM                   ;create par. quot. number

PQLOOP: rplvec  <#,PDIG>,r0,r1          ;(r1 is index to par. quot.)
        jmpngi  r3,DLENG2,DVLP1
        deci    r3
        deci    r1
        subvec  DIVDND,<#,PDIG>,r3
        ldivi   DDIG,r2,PDIG
        jmp     PQLOOP
;
; magnitude of par. quot. in r0, set sign

DVLP1:  gettyp  DIVDND,r3               ;sign of dividend
        movi    DIVSGN,r2
        andi    <#,F$SIGN>,r3
        xori    r3,r2                   ;xor with sign of divisor
        settyp  r2,r0                   ;set sign of quotient
        omovp   r0,TNUM                 ;store partial quotient
        movp    <#,TNUM>,r1
        movp    <#,QUOT>,r2
        call    ADDNUM                  ;add on partial quotient
        swapp   r0,QUOT                 ; to previous quotient

        movp    <#,TNUM>,r1
        movp    DIVSOR,r2
        call    MULNUM
        gettyp  r0,r1
        xori    <#,F$SIGN>,r1           ;flip sign for subtraction
        settyp  r1,r0
        swapp   r0,TNUM                 ;subtract prod. of partial quotient
        movp    <#,TNUM>,r1             ; & divisor from current dividend
        movp    <#,DIVDND>,r2
        call    ADDNUM                  ;to get new dividend
        subvec  r0,<#,DLENG1>           ;length of new dividend
        swapp   r0,DIVDND
        swapp   <#,NULL>,TNUM           ;dealc temp. product
        jmp     DVLOOP
;
; check that quotient & remainder are of same sign
; if not, decrement quotient and make remainder difference between divisor
; and original remainder

DVDONE: swapp   QUOT,r0
        jmpli   DLENG1,<#,2>,REMZER     ;remainder is zero
        gettyp  r0,r1                   ;sign of quotient
        gettyp  DIVDND,r3               ;sign of remainder
        jmpei   r1,r3,DVDON2

        call    DECMAG,DVDON2           ;decr. mag. of quotient
        cmovp   r0,QUOT
        xori    <#,F$SIGN>,r3           ;flip sign bit of remainder
        movp    DIVSOR,r2
        settyp  r3,<@,2>               ;divisor & remainder temporarily
        movp    <#,DIVDND>,r1           ; of opposite sign
        call    ADDNUM
        swapp   r0,DIVDND
        movp    DIVSOR,r0
        settyp  DIVSGN,<@,0>           ;restore sign of divisor
        cmovp   QUOT,r0
;
; remainder is zero--set sign to positive & return

REMZER: settyp  <#,F$NUM>,DIVDND

DVDON2: cmovp   DIVDND,r1
        pop     r3
        succeed
```

```
DIVZER: fail                            ;fail on divide check

        endmod Arith
```

4.2.2 BUFMGR.LIL

```
        search lil

        module bufmgr--Buffer management routines

        include FLAGS
        extrn   GIVEUP, EMPTY
;
;BUFFER CONTROL BLOCK FORMAT
;
;All routines in this module work off a buffer control block (BCB)
; which immediately precedes the buffer whose address in passed in R1

w1:     int     0               ;physical size of buffer(supplied by caller)
w2:     int     0               ;number of un(written,read) words in buffer
w3:     ptr     null            ;pointer to current word in buffer
w4:     ptr     null            ;On input,address of routine which supplies
                                ; a (possibly transient) pointer to the object
                                ; (supplied by caller)
                                ;On output,pointer to object (supplied by
                                ;caller, should be NULL if no object exists
                                ;on initialization)
w5:     int     0               ;offset in object to first (byte,word) in
                                ;next buffer to be (read,written). Supplied
                                ;by caller on input.
w6:     int     0               ;Length of object
w7:     int     0               ;BCB flags ( supplied by caller). B$DONE set
                                ; when input object exhausted. B$LAST set when
                                ; current buffer is last.
w8:;    blki                    ;start of buffer

        const BUFSIZ,w1-w8
        const BUFCNT,w2-w8
        const BUFPTR,w3-w8
        const OBJPTR,w4-w8
        const OBJCNT,w5-w8
        const OBJLNG,w6-w8
        const BUFLGS,w7-w8
        const BUFFER,w8-w8

        psect

;[
;routine: BININI -- initialize buffer control block for normal input
;input reg:r1--addr. of buffer
;output reg: destroys r0
;type: utility
;attr: SAFE
;calls: GETBUF
;]
        entry BININI

BININI: push    r2
        movp    <OBJPTR,1>,r2
        call    <@,2>                  ;get ptr. to object in r0
        getlen  r0,<OBJLNG,1>
        movp    r0,r2
        call    GETBUF                  ;load in 1st bufferful
        pop     r2
        return
;
;[
;routine: GETBUF -- loads next input buffer
;input regs:r1--addr of buffer, r2--ptr. to input object
;attr: SAFE
;]
        entry   GETBUF

GETBUF: tstb    <#,B$DONE>,<BUFLGS,1>,GIVEUP ;jump if end of object reached
        push    r3
        movp    r1,<BUFPTR,1>
;
;calculate size of transfer --either whole buffer or all remaining bytes

        movi    <OBJLNG,1>,r3
        subi    <OBJCNT,1>,r3           ;r3=remaining bytes - 1
        jmpli   r3,<BUFSIZ,1>,GETDON
        movi    <BUFSIZ,1>,r3
        jmp     GETB1

GETDON: inci    r3                      ;this is last transfer
        setb    <#,B$LAST>,<BUFLGS,1>

GETB1:  movi    r3,<BUFCNT,1>
        tstb    <#,B$STR>,<BUFLGS,1>,GETBS
        subvec  r2,r1,<OBJCNT,1>,r3,GIVEUP
        jmp     GETB2

GETBS:  substr  r2,r1,<OBJCNT,1>,r3,GIVEUP

GETB2:  addi    r3,<OBJCNT,1>
        pop     r3
        return

;[
;routine: RGTBUF — initializes buffer for reverse input (from end of object)
;                 OBJCNT is highest offset in object to be read
;                 FOR STRINGS ONLY
;input regs: r1— buffer ptr., r2—ptr. to object
;output reg: r0 destroyed
;attr: SAFE
;]
        entry   RGTBUF

RGTBUF: tstb    <#,B$DONE>,<BUFLGS,1>,GIVEUP
        push    r3
;
;calc. size of transfer— either whole buffer or remaining bytes

        movi    <BUFSIZ,1>,r3           ;r3 has size of transfer
        movi    <OBJCNT,1>,r0           ;r0 will have new OBJCNT
        subi    r3,r0
        jmppi   r0,RGTBU1               ;if positive, transfer whole buffer
                                        ; since more remains in object
; last buffer

        setb    <#,B$LAST>,<BUFLGS,1>
        movi    <#,0>,r0                ;no bytes left
        movi    <OBJCNT,1>,r3           ;transfer all remaining bytes
;
; transfer buffer & adjust BCB

RGTBU1: movi    r0,<OBJCNT,1>
        inci    r0                      ;r0 has offset of 1st byte in transfer
        substr  r2,r1,r0,r3,GIVEUP
        movi    r3,<BUFCNT,1>
        deci    r3
        cvtip   r3,r3
        addp    r1,r3                   ;r3 points to start of buffer
        movp    r3,<BUFPTR,1>
        pop     r3
        return

;
```

```
;[
;routine: BINP —get next (char,word) from buffered input object
;input reg: r1 —buffer addr.
;output reg: r0
;type: utility
;attr: calls GETBUF,SAFE
;]
        entry   BINP

BINP:   push    r2
        jmppi   <BUFCNT,1>,BIN1         ;jump if buffer not empty
        tstb    <#,B$LAST>,<BUFLGS,1>,BINDON
        movp    <OBJPTR,1>,r2
        call    <@,2>                   ;get ptr. to object
        movp    r0,r2
        call    GETBUF

BIN1:   movp    <BUFPTR,1>,r2
        tstb    <#,B$STR>,<BUFLGS,1>,BINS
        movp    <@,2>,r0
        jmp     BIN2

BINS:   movi    <@,2>,r0

BIN2:   incp    <BUFPTR,1>
        deci    <BUFCNT,1>
        pop     r2
        return
;
; come here if no more characters in object

BINDON: setb    <#,B$DONE>,<BUFLGS,1>
        movi    <#,0>,r0
        pop     r2
        return

;[
;routine:RBINP —get previous char from  buffered reverse input object
;input reg: r1 —buffer addr.
;output reg: r0
;type: utility
;attr: SAFE
;calls: GETBUF
;]
        entry   RBINP

RBINP:  push    r2
        jmppi   <BUFCNT,1>,RBIN1                ;jump if buffer not empty
        tstb    <#,B$LAST>,<BUFLGS,1>,BINDON     ;(in BINP)
        movp    <OBJPTR,1>,r2
        call    <@,2>                   ;get ptr. to object
        movp    r0,r2
        call    RGTBUF

RBIN1:  movp    <BUFPTR,1>,r2
        movi    <@,2>,r0
        deci    <BUFPTR,1>
        deci    <BUFCNT,1>
        pop     r2
        return

;
;routine: SETINP --resets input BCB to new offset within input object. Tries
;                to avoid reloading buffer
;input regs: r1- buffer ptr. r2-- new offset within object
;temp. regs: destroys r0
;attr: SAFE
;type: utility
;calls: BININI
;]
        entry SETINP

SETINP: push    r3
        clrb    <#,B$DONE>,<BUFLGS,1>    ;rejuvenate finshed buffers
;
; check if new offset is in current buffer
        movi    <OBJCNT,1>,r3            ;get offset of next buffer
        jmpnli  r2,r3,SETINI             ;jump if offset past current buffer
;
; get offset of start of buffer

        movp    <BUFPTR,1>,r0
        subp    r1,r0
        cvtpi   r0,r0                    ;R0 has current offset in buffer
        addi    <BUFCNT,1>,r0            ;"   "  size of current buffer
        subi    r0,r3                    ;R3 has offset of start of buffer
        jmpli   r2,r3,SETINI             ;jump if before current buffer
;
; offset in buffer, reset BCB

        subi    r3,r2                    ;get offset of new position in buffer
        movi    r0,<BUFCNT,1>
        subi    r2,<BUFCNT,1>
        cvtip   r2,r2
        addp    r1,r2                    ;calc. new pointer to buffer
        movp    r2,<BUFPTR,1>
        jmp     SETDON
;
; offset outside of buffer, re-initialize

SETINI: clrb    <#,B$LAST>,<BUFLGS,1>
        movi    r2,<OBJCNT,1>
        call    BININI

SETDON: pop     r3
        return

        subttl OUTPUT ROUTINES

;[
;routine: PUTBUF —concatenates object & buffer,creating new object.
;input reg: r1 (buffer addr.)
;type: utility,called by BOUP,BOUEND
;attr: UNSAFE
;]
        entry PUTBUF

PUTBUF: addi    <BUFCNT,1>,<OBJLNG,1>   ;calc. size of new object
        tstb    <#,B$STR>,<BUFLGS,1>,PUTSTR
;
;create new output vector.
;1)Create new object
;2)Copy current buffer to its end
;3)Copy old output vector in start of new vector,bufferful by bufferful

        vsect

OLDOBJ: ptr     NULL                    ;temp storage for ptr to old
                                        ; output object
        psect

PUTVEC: pushm   r2,r3
        swapp   <OBJPTR,1>,OLDOBJ       ;OLDOBJ owns old vector
        movp    r1,r3
        addp    <#,OBJPTR>,r3           ;r3 indirect ptr. to new vector
        movi    <#,F$VEC>,r2
        pushp   r1
        movi    <OBJLNG,1>,r1
        host    $ALLOC                  ;create new vector
        popp    r1
;
;copy buffer

        rplvec  r1,<OBJPTR,1>,<OBJCNT,1>,<BUFCNT,1>
```

```
;
;Copy old vector to new

        jmpli   <OBJCNT,1>,<#,1>,PUTV3  ;skip if no old object
        movi    <#,1>,r0                ;r0 is offset to start of next
                                        ;transfer
PUTV1:  movi    r0,r3                   ;r3 is offset to start of current
        addi    <BUFSIZ,1>,r0           ;transfer
        jmpnli  r0,<OBJCNT,1>,PUTV2     ;jump if not a full buffer
        subvec  OLDOBJ,r1,r3,<BUFSIZ,1> ;read old into buffer
        rplvec  r1,<OBJPTR,1>,r3,<BUFSIZ,1> ;write buffer into new
        jmp     PUTV1

PUTV2:  movi    <OBJCNT,1>,r0
        subi    r3,r0                   ;r0 is now size of last transfer
        jmpnpi  r0,PUTV3
        subvec  OLDOBJ,r1,r3,r0
        rplvec  r1,<OBJPTR,1>,r3,r0

PUTV3:  swapp   <#,NULL>,OLDOBJ         ;de-allocate old object
        popm    r2,r3
        jmp     PUTDON

PUTSTR: jmpzi   <OBJLNG,1>,PUTEMP
        movi    <#,NULL>,r0
        apps    r1,<OBJPTR,1>,r0,<BUFCNT,1> ;r0 gets new object
        swapp   r0,<OBJPTR,1>           ;(old object is de-allocated)
        jmp     PUTDON

PUTEMP: swapp   <#,EMPTY>,<OBJPTR,1>

PUTDON: movi    <#,0>,<BUFCNT,1>
        movi    <OBJLNG,1>,<OBJCNT,1>
        inci    <OBJCNT,1>
        movp    r1,<BUFPTR,1>
        return

;[
;routine: RPTBUF --writes out reverse output buffer
;                 for strings only
;input reg: r1 -- buffer addr.
;type : called by LAST
;attr: UNSAFE
;]
        entry   RPTBUF

RPTBUF: pushm   r2,r3
;
```

```
; create object for current buffer in <OBJPTR,1>

        omovp   <OBJPTR,1>,OLDOBJ
        movp    r1,r3
        pushp   r1
        movi    <OBJLNG,3>,r1
        addi    <BUFCNT,3>,r1           ; size of new object
        movi    r1,<OBJLNG,3>
        movi    <#,F$STR>,r2
        addp    <#,OBJPTR>,r3
        host    $ALLOC
        popp    r1
;
; write buffer,old object into new object

        movi    <BUFCNT,1>,r2           ;r2 has length of buffer
        rplstr  <BUFPTR,1>,<OBJPTR,1>,<#,1>,r2 ;write buffer
        inci    r2
        inss    OLDOBJ,<OBJPTR,1>,r2    ;write old object
        swapp   <#,NULL>,OLDOBJ         ;deallocate old object
;
; reset BCB & return

        popm    r2,r3
;
; RBOINI —entry point to initialize BCB —called from PREVLS, CVINS

        entry   RBOINI

RBOINI: movi    <#,0>,<BUFCNT,1>
        movi    <BUFSIZ,1>,r0
        cvtip   r0,r0
        addp    r1,r0
        movp    r0,<BUFPTR,1>           ;BUFPTR starts off at end of buffer
        return
;
;[
;routine: BOUINI —initialize output BCB
;input reg: r1 (addr. of buffer)
;type: utility
;attr: SAFE
;]
        entry BOUINI

BOUINI: movi    <#,0>,<BUFCNT,1>
        movp    r1,<BUFPTR,1>           ; BUFPTR points to start of buffer
        jmpep   <#,NULL>,<OBJPTR,1>,BOUIN1
        jmpep   <#,EMPTY>,<OBJPTR,1>,BOUIN1
        getlen  <OBJPTR,1>,<OBJLNG,1>
        movi    <OBJLNG,1>,<OBJCNT,1>
        inci    <OBJCNT,1>
        return

BOUIN1: movi    <#,0>,<OBJLNG,1>
        movi    <#,1>,<OBJCNT,1>
        return
;[
;routine: BOUP --writes (word,byte) in buffer
;input reg: r1-buffer, r2 -- input (word,byte)
;calls: PUTBUF
;type: utility
;attr: UNSAFE
;]
        entry   BOUP

BOUP:   jmpli   <BUFCNT,1>,<BUFSIZ,1>,BOUP1

        call    PUTBUF

BOUP1:  movp    <BUFPTR,1>,r0
        tstb    <#,B$STR>,<BUFLGS,1>,BOUS
        omovp   r2,<@,0>                ;place pointer in buffer
        jmp     BOUP2

BOUS:   movi    r2,<@,0>                ;place integer in buffer

BOUP2:  incp    <BUFPTR,1>
        inci    <BUFCNT,1>
        return

;[
;routine: RBOUP --writes byte in reverse output buffer
;input reg: r1-buffer, r2 -- input byte
;calls: RPTBUF
;type: utility
;attr: UNSAFE
;]
        entry   RBOUP

RBOUP:  jmpli   <BUFCNT,1>,<BUFSIZ,1>,RBOUP1
        call    RPTBUF
RBOUP1: decp    <BUFPTR,1>
        movp    <BUFPTR,1>,r0
        movi    r2,<@,0>
        inci    <BUFCNT,1>
        return
```

```
;
;[
;routine: BOUEND --closes up output buffer
;input reg: r1 -- buffer
;calls: PUTBUF
;type :utility
;attr: UNSAFE
;]
        entry   BOUEND

BOUEND: jmpzi   <BUFCNT,1>,BOUEN1
        call    PUTBUF

BOUEN1: setb    <$,B$DONE>,<BUFLGS,1>
        return

        endmod BUFMGR
```

4.2.3 CKBOOL.LIL

```
        search lil

        module CKBOOL -- checks if arbitrary string is a boolean value

        include FLAGS
        include CONSTS

; the boolean pseudo-strings

        pfix    F$STR,4
TRUEP:  text    "TRUE"

        pfix    F$STR,5
FALSP:  text    "FALSE"

        entry   TRUEP, FALSP

;[
;routine: CKBOOL -- fails if input is not boolean
;input regs: r1-- string ptr.
;output reg: r0--boolean string value, deallocates r1
;type :utility
;]
```

```
        entry   CKBOOL

CKBOOL: cmpsu   r1,<#,TRUEP>,CKBOO2,CKBOO2
        movp    <#,TRUEP>,r0
        jmp     CKGOOD

CKBOO2: cmpsu   r1,<#,FALSP>,CKFAIL,CKFAIL
        movp    <#,FALSP>,r0
CKGOOD: swapp   <#,NULL>,r1      ; deallocate literal only if no error
        succeed

CKFAIL: fail

        endmod  CKBOOL
```

4.2.4 EVAL.LIL

```
       search lil

        module EVALUATOR
        include FLAGS
        include CONSTS
        include ERRORS
;
;main evaluator for the Portable LOGO system
;called by the initializer
; register usage (in general, while a procedure is executing)
; R4 is the frame pointer & should be preserved by all routines
; R3 is a transient pointer to the current procedure directory
; R2 is a ptr. to the argument buffer when calling a primitive or
;       when setting up or removing local variables from the stack frame.
;    or an indirect pointer to the parse vector of the current line
;       when evaluating a user procedure.

; R1,R0 are temporaries

        vsect
;
;**********************************************************************
; EVAL stack frame integer variables
;  in order to save stack space, these are obtained or calulated from
;  the current procedure directory every time a stack frame
;  is created or restored
;  a call to EVINTS will do this nicely if CURPRC is set correctly
;  UNFRM does it automatically

NINTS:  int 0                          ;from procedure directory
NLOCS:  int 0                          ;number of locals
NARGS:  int 0                          ;number of arguments
NLINES: int 0                          ;number of lines
PFLAGS: int 0                          ;procedure directory flags
LASLIN: int 0                          ;offset in proc.dir. to last parse
                                       ;  vector
;**********************************************************************
;
; EVAL temporaries
;
NUMLEN: int 0                          ;length of number being copied
NUMFLG: int 0                          ;number's flags
SRCOFF: int 0                          ;offset in proc. dir. to source
                                       ; code of line to be parsed
OPCODE: ptr null                       ;current OM opcode
TRANS:  ptr null                       ;transient pointer
TRANS2: ptr null                       ; ditto
TRANS3: ptr null                       ; ditto
;
;**********************************************************************
;
; Buffer Control Block for use by LEXGET, MLXGET

EVBSIZ: int     EVBLEN
EVBCNT: int     0
EVBPTR: ptr     null
EVBOBJ: ptr     null
EVBOFF: int     0
EVBLNG: int     0
EVBFLG: int     B$STR                  ; strings always
EVBBUF: blki    EVBLEN
;
;**********************************************************************
;
; EVAL globals
        entry CURPRC, LASPRC, LEVEL, ERRCOD, ERRINF, ERROBJ, ERRCNT
        entry PRVUSR, $PI, GCNAME, TOPLEV

GCNAME: int 0                          ;non-zero if name table entries
                                       ; have been erased but not REMOVEd
PRVUSR: int 0                          ;non-zero if privileged user

$PI:    int 0                          ;polled interrupt word
```

```
LEVEL:  int 0                            ;no. of current stack frame

TOPLEV: int 0                            ;frame level of highest monitor
                                         ; (where name GC should take place)
CURPRC: ptr null                         ;pointer to PT entry of current proc.

LASPRC: ptr null                         ;  "      "  "   "    " last proc. or
                                         ;non-jump primitive (i.e., this
                                         ;tells who returned last value on the
                                         ;stack)
; ERROR HANDLER GLOBALS

ERROBJ: ptr null                 ;ptr. to object causing error
                                 ;(direct for parse errors,
                                 ; indirect for others)
ERRINF: ptr null                 ;Sentence describing context of the error
                                 ;( usually RUNnable)
ERRCOD: int 0                    ;Integer error code

ERRCNT: int 0                    ;position of ERROBJ within error contxt

        extrn LEXRCH                     ;contains addr. of routine to
                                         ;supply char. to scanner
        extrn NT                         ; addr. of Name Tablexpus

;
;external procedures
        extrn CKBOOL,ERRARG,LIST,PEXPR,PARSE,BININI,BINP,GIVEUP,TRPERR,TRPHDL
        extrn DECMAG
;
;
;       evaluator pseudo-strings found in LGODAT
;       used for building error contxts (ERRINF)

        extrn   IF,WHIL,UNTL,REPT        ;these are reserved words found

        extrn   RUNSTR,VALSTR            ;these are primitive names
        extrn   TRUEP,FALSP,EMPTY        ;special names
        extrn   THGSTR
;
;       Dispatch table for evaluation of OM code

        psect                            ;read-only

JMPTAB: ptr     LITC
        ptr     VALC
        ptr     DREF
        ptr     JUMP
        ptr     JMPF                    ;O$JMPI & O&JMPW both dispatch to the same
        ptr     JMPF                    ; address. (IF & WHILE statements)
        ptr     JMPT                    ;O$JMPU (UNTIL statement)
        ptr     DECJ
        ptr     NCAL
        ptr     EVEOL
        ptr     EOS

        const   OMJMP,JMPTAB-OMFAKE         ;converts an OM opcode to
                                            ; an index into JMPTAB
;
        entry   EVAL

; On entry to EVAL (and on subsequent iterations) CURPRC must contain
;  a ptr. to the symbol table entry of the procedure to be executed,
;  the previous stack frame must be set up correctly with TOP pointing
;  to the arguments on the local stack.

EVAL:   inci    LEVEL
        call    EVINTS                      ;set up NINTS thru LASLIN
        movi    NINTS,R0                    ;R0 will be offset in proc. dir. to
                                            ;to args. & locals
        movp    <TOP,4>,R2                  ;ptr. on stack to 1st arg.
        movi    NARGS,R1                    ;arg. count
;
;get new bindings for arguments (& save old)

GARGS:  jmpzi   R1,GLOCS
        inci    R0
        subvec  R3,<#,TRANS>,R0             ;TRANS gets NT entry of arg.
        getval  TRANS,TRANS2                ;TRANS2 is current binding
        omovp   TRANS2,TRANS3               ; make TRANS3 the owner
        setval  <@,2>,TRANS                ;make new binding for arg. from stack
        omovp   TRANS3,<@,2>               ;store former binding on stack
        decp    R2
        deci    R1
        jmp     GARGS

;
;save bindings & undefine local variables

GLOCS:  movi    NLOCS,r1                    ;count of LOCALS
GLOCS2: jmpzi   r1,SETFRM
        inci    r0
        subvec  r3,<#,TRANS>,r0             ;NT entry of local
        getval  TRANS,TRANS2                ;current binding
        omovp   TRANS2,<@,2>               ;place binding on stack
        setval  <#,UNDEF>,TRANS             ;and unbind
        decp    r2
        deci    rl
        jmp     GLOCS2

;
;create constant part of stack frame

SETFRM: incp    r2
        subp    <#,FRMSIZ>,r2           ;r2 is (temporarily) new frame ptr.
        movp    r4,<PREV,2>             ;set pointer to old frame
        movp    r2,r4                   ;r4 returns as FRAME ptr.
        movp    CURPRC,<PROC,4>
;
;find privileges of frame

        movi    <#,EV$PRV>,<EVFLGS,4>
        jmpnzi  PRVUSR,SETFRl
        getflg  CURPRC,r0
        tstb    <#,F$PRIV>,r0,SETFRl
        movi    <#,0>,<EVFLGS,4>        ;no flags on

SETFRl: movi    NINTS,rl                ;calc. offset within proc. dir.
        addi    NLOCS,rl
        addi    NARGS,rl                ;to first parsed line.
        movi    rl,<LINE,4>

        movp    r4,<TOP,4>              ;initialize local stack ptr.
        movp    <#,TRANS>,r2            ;r2 is pointer to parse vector ptr.
;
; increment line no.

NEWLIN: inci    <LINE,4>
;
;       check if done with procedure

        jmpgi   <LINE,4>,LASLIN,DONE    ;equivalent to saying "DONE" in LOGO

;
;store transient pointer to current parse vector in TRANS

SETLIN: subvec  r3,<#,TRANS>,<LINE,4>
;
;check if line needs parsing

        jmpnep  TRANS,<#,NULL>,STLIN    ;NULL parse vector means no parse done
;
;parse source code for procedure
```

```
GETPRS: movi    <LINE,4>,SRCOFF          ; SRCOFF is offset within proc. dir
        addi    NLINES,SRCOFF            ;  to source code string
;
; set up BCB for getting source code via buffer management

        movp    <#,LEXGET>,EVBOBJ        ; addr. of routine to get string ptr.
        movi    <#,B$STR>,EVBFLG
        movi    <#,1>,EVBOFF             ; Offset to start of string
        movp    <#,EVBBUF>,r1            ; give addr. of buffer to buf. mgr.
        call    BININI                   ; init. buffer
        movp    <#,LXRCH>,LEXRCH         ;arg. to parser( char routine)
        call    PARSE,UPRERR
        getval  CURPRC,r3                ; restore r3
        movp    r0,TRANS                 ; store parse vector in transient &
        rplvec  <#,TRANS>,r3,<LINE,4>    ; in proc. dir
        jmp     STLIN                    ; start executing line
;
; user procedure parse error

UPRERR: call    LEXGET                   ;get ptr. to source string
        gettyp  r0,r1
        movs    r0,ERRINF                ;copy failed string as context
        settyp  r1,ERRINF
        decp    <TOP,4>
        movi    <#,0>,<PSOFF,4>          ;signal that line is not parsed
        jmp     ERRHDL                   ; on return from error trap
;
;
;[
;routine: VALOF (VALUEOF)
; parses & executes a string representing a LOGO expression (returns a
; value.  Causes RUNPRC to enter the parser at PEXPR
; Current privilege mode is maintained
; input reg: r1-(ignored), r2--indirect ptr. to string
; output reg:r1-stack frame flags,r3-parser entry point
;]
        entry   VALOF

VALOF:  movi    <EVFLGS,4>,r1            ;get current flags
        andi    <#,EV$PRV>,r1            ;keep priv. mode flag only
        iori    <#,EV$VAL>,r1            ;add in VALUE flag
        movp    <#,PEXPR>,r3
        jmp     RUNPRC
;[
;routines: RUN, #PRIVRUN
;RUN parses and executes a full LOGO command line (including multiple
; commands).  It takes standard primitive args (1 word arg. buffer)
; & falls into RUNPRC. Causes RUNPRC to enter the parser at PARSE.
;If the user is not privileged, RUN turns off privileged mode.
;
;#PRIVRUN is the same except that privileges are turned on
; input regs: r1--(ignored), r2--indirect ptr. to string
; output reg:r1-stack frame flags,r3-parser entry point
;]
        entry   RUN, PRVRUN

PRVRUN: movi    <#,EV$PRN>,r1
        jmp     RUN1

RUN:    movi    <#,EV$RUN>,r1           ;flags for RUNPRC
RUN1:   movp    <#,PARSE>,r3
;[
;RUNPRC--creates special stack frame,falls into evaluator
;input regs: r1--flags for EVFLGS entry in stack frame
;            r2--indirect ptr. to source string
;            r3--parser entry point
;]
        entry   RUNPRC

RUNPRC: inci    LEVEL
        movp    r4,r0
        movp    <TOP,4>,r4
        subp    <#,FRMSIZ>,r4           ;change frame ptr.
        movp    r0,<PREV,4>             ;save old frame ptr.
        movp    r4,<TOP,4>              ;init. local stack ptr.
        movp    <#,NULL>,<PROC,4>       ;mark this as RUN frame
        movp    <#,NULL>,CURPRC         ;ditto
        jmpzi   PRVUSR,RUNPR2           ;mark RUN frame as privileged
        setb    <#,EV$PRV>,r1           ; ONLY for priv. users

RUNPR2: movi    r1,<EVFLGS,4>           ;store flags
        cmovp   <@,2>,<LINE,4>         ;store source code on stack
;
;now set up for parse

        movi    <#,B$STR>,EVBFLG
        movi    <#,1>,EVBOFF
        movp    <#,MLXGET>,EVBOBJ
        movp    <#,EVBBUF>,r1           ; give addr. of buffer to buf. mgr.
        call    BININI
        movp    <#,LXRCH>,LEXRCH        ;arg. to parser
        call    <@,3>,MPRERR           ;call parser
        swapp   r0,<LINE,4>             ; store parse vector on stack
        movp    r4,r2
        addp    <#,LINE>,r2             ; r2 points to parse vector
        jmp     STLIN
```

```
;
; Run procedure parse error

MPRERR: movi    <#,0>,<PSOFF,4>         ;signal this is parse error
        decp    <TOP,4>
        swapp   <LINE,4>,ERRINF         ;get failed string as context
        jmp     ERRHDL                  ; on return from error trap
;
; Main evaluation loop
; begin execution of parsed line

STLIN:  movi    <#,2>,<PSOFF,4>         ; offset to first instruction in
                                        ; parse vector
;
;NXTINS--restores proc. dir. transient & parse vector transient
;                                       (for non-run procedures)

NXTINS: host    $POLL                   ;sync. with operating system
                                        ;(storage allocator possibly called)
        jmpep   <#,NULL>,CURPRC,GETINS
        getval  CURPRC,r3
        subvec  r3,<#,TRANS>,<LINE,4>
;
; then get next instruction

GETINS: jmpnzi  $PI,EVLTRP              ;check for interrupts
        subvec  <@,2>,<#,OPCODE>,<PSOFF,4>; get opcode
        inci    <PSOFF,4>               ;incr. PC
        movp    OPCODE,r0
        movp    <OMJMP,0>,r0            ;turn OPCODE into index to dispatch
        jmp     <@,0>                  ; table & jump
;
; LITC--Stack literal

LITC:   subvec  <@,2>,<#,TRANS3>,<PSOFF,4>; get literal
        inci    <PSOFF,4>
        decp    <TOP,4>                 ; increment(sic) local stack ptr.
        movp    <TOP,4>,r0              ; get it
        movp    <#,NULL>,<@,0>         ;clear junk on stack
;
;all copying onto stack is done here

STKCOP: jmpep   TRANS3,<#,EMPTY>,STKEMP
        gettyp  TRANS3,r1
        tstb    <#,F$NUM>,r1,STKCP1
        movs    TRANS3,<@,0>           ; copy onto stack(Transients clobbered)
        jmp     STKCP2
STKCP1: getlen  TRANS3,NUMLEN
```

```
        extn    TRANS3,<@,0>,<#,1>,NUMLEN

STKCP2: settyp  r1,<@,0>                ; set flags
        jmp     NXTINS
;
; STKEMP-- stack empty object

STKEMP: swapp   <#,empty>,<@,0>                  ;(r0 is local stack stack pointer)
        jmp     nxtins
;
; VALC--stack value of literal
;       (literal has already been converted to symbol table ptr.
;        by parser)

VALC:   decp    <TOP,4>                 ; incr. stack ptr.
        movp    <TOP,4>,r0
        subvec  <@,2>,<#,TRANS2>,<PSOFF,4>; get symbol table ptr.
        inci    <PSOFF,4>
        movp    <#,NULL>,<@,0>         ;clear junk on stack
        jmp     evlref                  ; replace with value using code
                                        ;  from DREF
;
;THING -- uses DREF

        entry   THING

THING:  call    SETR2                   ;restore R2 as code ptr.
        movi    <#,O$DREF>,OPCODE       ;so REFERR will know
;
; DREF--replace top of stack with its value

DREF:   movp    <top,4>,r0
        gettyp  <@,0>,r1               ;check if name is word
        tstb    <#,f$list>,r1,tolerr    ; error if not
        find    nt,<@,0>,trans2                 ; trans2 gets symbol table entry
        jmpep   trans2,<#,null>,UDFERR  ;error if undefined name
EVLREF: getflg  TRANS2,r1
        tstb    <#,F$NRD>,r1,RDERR      ;no read
        tstb    <#,F$TRCE>,r1,RDTRC     ;trace trap
        getval  trans2,trans3
        jmpep   trans3,<#,undef>,UDFERR ;error if value is undefined
        jmp     STKCOP                  ;else go stack value
;
; 'thing of a list' error

tolerr: movi    <#,e$welf>,errcod       ; "word expected, list found"
        jmp     REFER2
;
```

```
; undefined name error or priv. violation or trace trap

RDERR:  movi    <#,E$OBPV>,ERRCOD
        jmp     REFERR

RDTRC:  movi    <#,E$TRCR>,ERRCOD
        jmp     REFERR

UDFERR: movi    <#,E$UNDF>,ERRCOD
REFERR: jmpep   <#,O$DREF>,OPCODE,REFER2 ; valc or dref?
                                        ;valc-undefined name is in name table
        getnam  TRANS2,TRANS3           ; entry referenced by TRANS2
        movs    TRANS3,<@,0>           ;store variable name on stack

REFER2: movi    <#,2>,ERRCNT
        movp    <#,THGSTR>,r3
        movp    r0,r2                   ;indirect ptr. to error object
        movi    <#,1>,r1
        call    ERRARG,GIVEUP           ;[ : [ name ] ] is error context
        swapp   r0,ERRINF
        jmp     ERRHDL
;
;JUMP—change OM pc

JUMP:   subvec  <@,2>,<#,TRANS2>,<PSOFF,4> ;TRANS2 is fake integer offset
SETPC:  movi    TRANS2,<PSOFF,4>        ; to parse vector
        subi    <#,NUMFAK>,<PSOFF,4>    ;turn fake int. to true integer
        jmp     NXTINS
;
;JMPT—change OM pc if top of stack is TRUEP or "TRUE
;       used for both IF (O$JMPI) & WHILE (O$JMPW)

JMPT:   subvec  <@,2>,<#,TRANS2>,<PSOFF,4>
        inci    <PSOFF,4>
        movp    <TOP,4>,r0
        incp    <TOP,4>                 ; pop stack
        jmpep   <@,0>,<#,TRUEP>,SETPC
        jmpep   <@,0>,<#,FALSP>,NXTINS
        omovp   <@,0>,r1
        call    CKBOOL,CKERR            ; convert literal to boolean ptr.
                                        ; take error jump if not boolean
                                        ; de-allocate literal if no error
        jmpep   r0,<#,TRUEP>,SETPC
        jmp     NXTINS

;
;JMPF—change OM pc if top of stack is FALSP or "FALSE
```

```
JMPF:   subvec  <@,2>,<#,TRANS2>,<PSOFF,4>
        inci    <PSOFF,4>
        movp    <TOP,4>,r0
        incp    <TOP,4>                 ; pop stack
        jmpep   <@,0>,<#,FALSP>,SETPC
        jmpep   <@,0>,<#,TRUEP>,NXTINS
        cmovp   <@,0>,r1
        call    CKBOOL,CKERR            ; convert literal to boolean ptr.
                                        ; take error jump if not boolean
                                        ; CKBOOL de-allocates literal if
                                        ;  no error
        jmpep   r0,<#,FALSP>,SETPC
        jmp     NXTINS
;
; non-boolean input to a conditional

CKERR:  movi    <#,E$NBIC>,ERRCOD
        movi    <#,2>,ERRCNT
        swapp   r1,ERROBJ
        jmpnep  <#,O$JMPI>,OPCODE,CKERR2;find out if conditional was
                                        ; IF, WHILE, or UNTIL
        movp    <#,IF>,r3
        jmp     CKERR4

CKERR2: jmpnep  <#,O$JMPW>,OPCODE,CKERR3
        movp    <#,WHIL>,r3
        jmp     CKERR4

CKERR3: movp    <#,UNTL>,r3

CKERR4: movp    <#,ERROBJ>,r2           ; make list  of conditional name
        movi    <#,1>,r1                ; & bad input for ERRINF
        call    ERRARG,GIVEUP
        swapp   r0,ERRINF
        swapp   <#,NULL>,ERROBJ         ;deallocate ERROBJ
        decp    <TOP,4>                 ;stack ptr. points to where new
                                        ; input to conditional should be
                                        ; returned by error trap
        subi    <#,2>,<PSOFF,4>         ;restore OM PC to retry conditional
                                        ;on return from error trap
        jmp     ERRHDL
;
;DECJ--jump if top of stack is not positive,else decrement

DECJ:   subvec  <@,2>,<#,TRANS2>,<PSOFF,4>
        inci    <PSOFF,4>
        movp    <TOP,4>,r1
        gettyp  <@,1>,NUMFLG           ;get flags of top of stack
```

```
;
;if not number, convert

        tstb    <#,F$NUM>,NUMFLG,DECJ1
        omovp   <@,1>,r1
        call    CVTSN,DECERR            ;put number in r0
        swapp   <#,NULL>,r1             ;dealc. old string
        movp    <TOP,4>,r1
        jmp     DECJ2
;
;check number & decrement if necessary

DECJ1:  omovp   <@,1>,r0
DECJ2:  tstb    <#,F$SIGN>,NUMFLG,DECJMP ;if number is negative,jump
        call    DECMAG,DECJMP           ;else try to decr., jump if number
                                        ; was zero
        swapp   r0,<@,1>               ;else restore number to top of stack
        jmp     NXTINS


DECJMP: swapp   <#,NULL>,r0             ; deallocate number
        incp    <TOP,4>                 ; pop stack
        jmp     SETPC

;
;non-numeric arg. to REPEAT (DECJ)

DECERR: movi    <#,E$NENF>,ERRCOD       ;"Number Expected, Not Found"
        movi    <#,2>,ERRCNT
        swapp   r1,ERROBJ
        movi    <#,1>,r1
        movp    <#,ERROBJ>,r2
        movp    <#,REPT>,r3             ;REPEAT pseudo-string
        call    ERRARG,GIVEUP
        swapp   r0,ERRINF               ; [ REPEAT arg ] is error context
        swapp   <#,NULL>,ERROBJ         ; dealc. copy of arg in TRANS2
        subi    <#,2>,<PSOFF,4>         ; allow retry of OM instr.
        jmp     ERRHDL
;
;procedure call-either user or primitive

NCAL:   subvec  <@,2>,<#,TRANS>,<PSOFF,4>,<#,2> ; pick up PT entry in TRANS,
                                        ; no. of args in TRANS2(phony ptr)
        addi    <#,2>,<PSOFF,4>

        getflg  TRANS,r0                ; r0 has proc. flags
        movp    TRANS2,r1
        subi    <#,NUMFAK>,r1           ; r1 has actual # of args. passed
```

```
;
; adjust local stack ptr. by (# of args)-1 words

        cvtip   r1,r3                   ; r3 is amount to
        decp    r3                      ; adjust local stack ptr.
        movp    <TOP,4>,r2
        addp    r3,r2                   ; r2 points to start of arg. buffer
;
;check for traps, greediness & primitive calls
;kludge--if this is a trapreturn, don't trace trap

        tstb    <#,EV$TRT>,<EVFLGS,4>,NCAL2
        tstb    <#,F$TRCE>,r0,PTRCE     ;check for trace trap
NCAL2:  tstb    <#,F$PRIM>,r0,PCAL

        movp    r2,<TOP,4>              ;adjust stack pointer for user call
        tstb    <#,F$GRDY>,r0,GCAL
        jmp     UCALL
;
;GCAL--handle greedy calls to user procs.
; make the args. to a user greedy into a list

GCAL:   push    r0                      ;save flags
        call    CVTSTR,GIVEUP           ;convert buffer to all strings
        pushm   r1,r2
        call    LIST,GIVEUP
        popm    r1,r2
        call    UNARG                   ; de-allocate args. to LIST
        movp    <TOP,4>,r1
        omovp   r0,<@,1>               ;move list of args onto stack
        pop     r0                      ;fall into UCALL
;
;user procedure

UCALL:  getval  TRANS,r3                ;(SEE ABOUT COMBINING WITH EVINTS)
        jmpep   r3,<#,UNDEF>,ERSERR     ;attempt to execute erased procedure
        tstb    <#,EV$PRV>,<EVFLGS,4>,UCALL2
        tstb    <#,F$PRIV>,r0,UCALL1
        jmp     UCALL2
UCALL1: gettyp  r3,r0
        tstb    <#,F$PINV>,r0,PRVERR    ;can't run priv.,invisible proc.

UCALL2: movp    TRANS,CURPRC            ; set PT entry of current procedure
        jmp     EVAL                    ; & start it up
;
;call primitive
;(r1 already has #. of args,
; r2 points to start (highest address) of arg. buffer)
```

```
PCAL:   tstb    <#,EV$PRV>,<EVFLGS,4>,PCAL1
        tstb    <#,F$PRIV>,r0,PRVERR    ;proc. privilege violation

PCAL1:  movp    TRANS,LASPRC             ; store PT entry of primitive
        call    CVTARG,CVTFAI            ;convert args. to strings or numbers
                                         ;as required (regs preserved)
        getval  TRANS,r3                 ; r3 has addr. of primitive
        tstb    <#,F$JMP>,r0,<@,3>      ; jump to primitive
        pushm   r1,r2
        pcall   <@,3>,PFAIL             ; or call primitive
        popm    r1,r2
        movp    r2,<TOP,4>               ;adj. stack pointer
        call    UNARG                    ; de-allocate args. to prim.
        movp    <TOP,4>,r1
        omovp   r0,<@,1>                ;move result of prim. onto stack
        jmp     GETRST                   ;go check result of prim. & continue
                                         ; with proc. if OK

;
;trace trap on procedure call

PTRCE:  movi    <#,E$TRCP>,ERRCOD
        movi    <#,1>,ERRCNT
        getnam  TRANS,r0
        movs    r0,r3                    ;copy procedure name
        pushm   r1,r2
        call    ERRARG,GIVEUP            ;make proc. name,args. into list
        swapp   r0,ERRINF
        popm    r1,r2
        movp    r2,<TOP,4>               ;adj. stack pointer
        call    UNARG                    ;dealc. args
        movi    <#,T$TRCE>,r1
        call    TRPERR,TRPCNT            ;call trap handler
        jmp     EVAL
;
;attempt to execute erased procedure

ERSERR: movi    <#,E$NXOB>,ERRCOD
        jmp     UCLERR
;
;priv. violation on procedure call

PRVERR: movi    <#,E$OBPV>,ERRCOD

UCLERR: movi    <#,1>,ERRCNT
        getnam  TRANS,r0
        movs    r0,r3                    ;copy procedure name
        jmp     PFAIL2
```

```
;
; arg. conversion or primitive failed
;  ERRCOD,ERROBJ supplied by primitive or PCAL
;  ERROBJ has indirect ptr. to offending arg.

PFAIL:  popm    r1,r2                   ;get back ptr. to arg. buffer
;
; find position of ERROBJ within arg. buffer

CVTFAI: movp    r2,r0
        subp    ERROBJ,r0
        cvtpi   r0,ERRCNT
        addi    <#,2>,ERRCNT
;
;build error context

        getnam  TRANS,r3                ;get name (always pseudo-heapject
                                        ; so no need to copy)
PFAIL2: pushm   r1,r2
        call    ERRARG,GIVEUP           ;make proc. name,args. into list
        swapp   r0,ERRINF
        popm    r1,r2
        movp    r2,<TOP,4>              ;adj. stack pointer
        call    UNARG                   ;dealc. args
        jmpzi   $PI,ERRHDL
        movi    <#,0>,ERRCOD            ;interrupt detected by primitive
        call    TRPHDL,TRPCNT           ;call trap handler
        jmp     EVAL
;
;EVEOL—end of line

EVEOL:  jmpnep  CURPRC,<#,NULL>,NEWLIN  ; get next line (if normal procedure)
                                        ;  else terminate RUN procedure
        tstb    <#,EV$VAL>,<EVFLGS,4>,RTURN1 ;if this is VALUE or TRAPRETURN
        tstb    <#,EV$TRT>,<EVFLGS,4>,RTURN1 ; then execute implicit RETURN
;
; there is a difference between an EVEOL & an explicit DONE or RETURN for RUN
; procedures, so just remove RUN stack frame & enter DONE past RUN loop

        call    UNFRM
        jmp     RDONE
;
;EOS—end of statement

EOS:    movp    <TOP,4>,r0              ; check if top of stack is NO RESULT
        jmpnep  <@,0>,<#,UNDEF>,EOSERR
        incp    <TOP,4>                 ;pop stack
        jmp     NXTINS
```

```
;
; operation found where command should be

EOSERR: movi    <#,E$OIC>,ERRCOD          ;"Operation Instead of Command"
        movi    <#,1>,ERRCNT
        swapp   <@,0>,ERRINF              ;value of operation
        deci    <PSOFF,4>                 ;allow retry of instruction
        jmp     ERRHDL

;
;GETRST checks if result of a proc. or primitive was undefined(i.e.,command)
;traps TRACEd returns
; if so, next OM instruction must be an EOS
; first restores transients & R2 as indirect parse ptr.
; assumes result is on top of local stack

        entry   GETRST                    ;(entered from $BOMB)

GETRST: jmpzi   <PSOFF,4>,EVEOL           ;if this is return from parse error
                                          ;then line has been retried already
                                          ;else continue with execution of line
        call    SETR2                     ;restore r2 as code ptr.
        tstb    <#,EV$TRT>,<EVFLGS,4>,NXTINS ;don't check result if TRAPRTRN
        movp    <TOP,4>,r0                ;if top of stack is not UNDEF
        getflg  LASPRC,r1
        tstb    <#,F$TRCE>,r1,GTRACE      ;check for trace trap

        jmpnep  <@,0>,<#,UNDEF>,NXTINS ; continue with next instruction
        subvec  <@,2>,<#,OPCODE>,<PSOFF,4>; otherwise, get next OM instr.
        inci    <PSOFF,4>
        jmpep   OPCODE,<#,O$EOL>,EVEOL
        jmpnep  OPCODE,<#,O$EOS>,CMDERR
        incp    <TOP,4>                   ;if no error, then pop stack &
        jmp     NXTINS                    ; continue

GTRACE: movi    <#,E$TRCF>,ERRCOD
        swapp   <@,0>,ERROBJ              ;result becomes trace trap object
        movi    <#,0>,ERRCNT              ;signal error object supplied
        jmp     CMDER1                    ;copy procedure name
;
;CMDERR -- command found where operation expected

CMDERR: movi    <#,E$CIO>,ERRCOD
        movi    <#,1>,ERRCNT
CMDER1: getnam  LASPRC,TRANS2
        movs    TRANS2,ERRINF             ;give name of command as context
        jmp     ERRHDL
;
```

```
;
;EVAL ERROR & INTERRUPT HANDLING

        entry   ERRHDL

EVLTRP: movi    <#,1>,ERRCOD            ;signal evaluator detected interrupt
        movp    <TOP,4>,r0
        jmpnep  r0,r4,EVLTR2            ;check if nothing's on stack
        decp    r0                      ;if so,place UNDEF on stack
        movp    r0,<TOP,4>
        movp    <#,UNDEF>,<@,0>

EVLTR2: swapp   <@,0>,ERRINF            ;set top of stack value as trap context
        call    TRPHDL,EVLTR3            ;just continue if interrupt can't be
        jmp     EVAL                     ; handled

EVLTR3: movp    <TOP,4>,r0
        swapp   ERRINF,<@,0>            ;restore err context to stacktop
        jmp     NXTINS

ERRHDL: movi    ERRCOD,r1
        quoi    <#,100>,r1
        inci    r1                      ;r1 is error type
        call    TRPERR,TRPCNT
        jmp     EVAL
;
; for primitive calls--
; if trap cannot be handled, simulate a TRAPRETURN on the error context

TRPCNT: movp    <TOP,4>,r2
        swapp   ERRINF,<@,2>            ;put context on stack
        movi    <#,EV$TRT>,r1            ;get stack frame flags
        movp    <#,PEXPR>,r3             ;parser entry point
        jmp     RUNPRC
;
;[
;ROUTINE: CVTARG -- converts argument buffer to strings or numbers
;               as required by called primitive
;input REGS: r0--proc. entry flags of called primitive
;               r1--# of args.,r2--top of arg. buffer,
;output REG: r0,r1,r2 preserved, fails if required conversion cannot be done
;ATTR:UNSAFE
;TYPE: called from PCAL
;       has global entry points CVTSTR,CVTNUM
;CALLS:
        extrn CVTNS,CVTSN
;]
        vsect
```

```
CERFLG: int     0                       ;non-zero if numeric conversions
                                        ; should return ERROBJ

        psect

CVTARG: tstb    <#,F$SARG>,r0,CVTSTR
        tstb    <#,F$NARG>,r0,CVTNU0
        succeed                         ;no conversion required

        entry   CVTSTR
;
;convert buffer to all strings

CVTSTR: pushm   r0,r3
        movi    r1,r3                   ;r3 counts args.
CVTST1: jmpzi   r3,CVTDON
        gettyp  <@,2>,r0
        tstb    <#,F$STR>,r0,CVTST2
        omovp   <@,2>,r1               ;get numeric arg.
        call    CVTNS                   ;r1 destroyed,string in r0
        swapp   r0,<@,2>
CVTST2: deci    r3
        decp    r2
        jmp     CVTST1

        entry   CVTNUM, CVTNU0
;
;convert buffer to all numbers (fail if not possible)

CVTNUM: movi    <#,0>,CERFLG            ;don't set ERROBJ on errors
        jmp     CVTNU1

CVTNU0: movi    <#,1>,CERFLG            ;set ERROBJ on errors

CVTNU1: pushm   r0,r3
        movi    r1,r3
CVTNU2: jmpzi   r3,CVTDON
        gettyp  <@,2>,r0
        tstb    <#,F$NUM>,r0,CVTNU3
        omovp   <@,2>,r1
        call    CVTSN,CVTNER
        swapp   r0,<@,2>
        swapp   <#,NULL>,r1             ;dealc. string
CVTNU3: deci    r3
        decp    r2
        jmp     CVTNU2
;
;success for both kinds of conversion
```

```
CVTDON: popm    r0,r3
        succeed
;
;numeric conversion error

CVTNER: jmpzi   CERFLG,CVTNE1
        movi    <#,E$NENF>,ERRCOD
        movp    r2,ERROBJ
CVTNE1: swapp   r1,<@,2>                ;put back arg.
        popm    r0,r3
        fail

;
;       some EVALUATOR utility routines
;[
;routine: SETR2 --restores R2 as code ptr.
;called by: THING, GETRT
;]

SETR2:  jmpnep  CURPRC,<#,NULL>,SETR2A
        movp    r4,r2                   ; r2 for RUN procs. points to LINE
        addp    <#,LINE>,r2
        return

SETR2A: movp    <#,TRANS>,r2            ; R2 for normal procedures
        getval  CURPRC,r3               ;R3 is proc. dir.
        subvec  r3,<#,TRANS>,<LINE,4>   ;TRANS gets parse vector
        return
```

```
;[
;routine:UNARG --de-allocates objects in argument buffer
;input regs:r1 (# of args),r2 (addr. of start of buffer)
;type: utility
;]

        entry   UNARG

UNARG:  jmpnpi  r1,UNARG2
        swapp   <#,NULL>,<@,2>          ; implicit de-allocation
        deci    r1
        decp    r2
        jmp     UNARG

UNARG2: return

;[
;routine: UNSTK --deallocates local stack of current stack frame
;type: utility
;uses: r0
;]
        entry   UNSTK

UNSTK:  movp    <TOP,4>,r0
UNSTK1: jmpngp  r4,r0,UNSTK2             ;r4,as frame ptr,points to word above
                                         ; bottom of local stack
        swapp   <#,NULL>,<@,0>          ;deallocate word on stack
        incp    r0
        jmp     UNSTK1

UNSTK2: return

;[
;routine: UNFRM--undoes current stack frame
;type : utility
;regs: affects all registers
;globals: r4 (frame pointer),CURPRC
;         NINTS,NARGS,NLOCS,NLINES
;calls: UNSTK
        extrn DISTRP, GARBAG
;attr: de-allocates & swapps
;]

        entry   UNFRM
;
;first,de-allocate objects on local stack frame

UNFRM:  deci    LEVEL
        call    UNSTK
;
;if frame is RUN proc.,merely de-allocate parse vector on stack
```

```
UNFRM2: jmpnep  <#,NULL>,CURPRC,UNFRM3
        swapp   <#,NULL>,<LINE,4>
        movp    <PREV,4>,r4              ; r4 becomes previous frame ptr.
        jmp     UNFRM5                   ;skip restoration of locals
;
;otherwise,restore bindings of local variables from stack

UNFRM3: movp    CURPRC,LASPRC            ;remember this procedure name
                                         ; in case it returned a bad value
        movp    <PREV,4>,r4              ;r4 becomes previous frame ptr.
        getval  CURPRC,r3                ;r3 =(trans) ptr. to proc.dir.
        movp    <TOP,4>,r2               ;prev. frame's top of stack
        movi    NINTS,r0                 ;offset to vars. in proc. dir.
        movi    NARGS,r1
        addi    NLOCS,r1                 ;no. of vars. local to proc.
;
;move thru stack,replacing stored bindings
UNFRM4: jmpzi   r1,UNFRM5
        inci    r0
        subvec  r3,<#,TRANS>,r0          ;get NT entry for var.
        setval  <@,2>,TRANS             ;restore binding(transfers ownership)
        decp    r2                       ; so implicitly de-allocates from stack
        deci    r1
        jmp     UNFRM4
;
;now frame is undone, check for traps, garbage,& restore EVAL globals

UNFRM5: movi    <EVFLGS,4>,r1            ;check if this frame was trap
        andi    <#,EV$PI>,r1             ; recipient
        jmpzi   r1,UNFRM6                ;if PI part of flag word was non-zero
        call    DISTRP                   ; then dismiss the trap

UNFRM6: jmpzi   GCNAME,UNFR6A            ;no garbage extant
        jmpnei  TOPLEV,LEVEL,UNFR6A      ;only collect at toplevel frame
        call    GARBAG
        movi    <#,0>,GCNAME

UNFR6A: movp    <PROC,4>,CURPRC
        jmpep   CURPRC,<#,NULL>,UNFRM7   ;restore globals only if not a
        call    EVINTS
        movp    <#,NULL>,r3              ;destroy transient ptr. output
                                         ; of EVINTS
UNFRM7: return

;[
;routine: UNWIND
;input reg: r1--level of stack frame. Unwind to this frame.
;output reg: r4 adjusted
;type: utility
;calls: UNFRM
;attr: SAFE
;]
        entry   UNWIND

UNWIND: pushm   r2,r3
UNWIN1: jmpngi  LEVEL,r1,UNWIN2         ;this should terminate on equality
        pushi   r1
        call    UNFRM                   ;r4 adjusted
        popi    r1
        jmp     UNWIN1

UNWIN2: popm    r2,r3
        return

;[
;routine:EVINTS
;output reg: r3--transient ptr. to proc. dir.
;globals:NINTS thru LASLIN, CURPRC
;]

EVINTS: getval  CURPRC,R3               ;R3 is trans.ptr. to proc. dir.
        subvec  r3,<#,NINTS>,<#,1>,<#,4>;restore NINTS thru NLINES
        gettyp  r3,PFLAGS               ;get procedure flags
;
;calc. offset in proc. dir. to last parse vector

        getlen  r3,LASLIN
        subi    NLINES,LASLIN

EVINT2: return

;
;[
;routine: GO--assumes line numbers are 1 digit (not greater than BASE1)
;attr: Jump primitive
;globals:
        extrn   GOSTR
;calls: ERRARG
;]
        vsect

LINENO: int     0
GOPRV:  int     0                       ;non-zero if GO invoked in
                                        ; priv. mode
        psect
        entry   GO

GO:     movp    <TOP,4>,r0
        gettyp  <@,0>,r1
        tstb    <#,F$SIGN>,r1,NOGO1      ;no negative numbers
        subvec  <@,0>,<#,NUMLEN>        ;get length of number
        jmpnei  NUMLEN,<#,2>,NOGO1       ;jump if zero or too big
        subvec  <@,0>,<#,LINENO>,<#,2> ;get actual number
;
; test if in privileged mode

        movi    <#,1>,GOPRV
        tstb    <#,EV$PRV>,<EVFLGS,4>,GO1
        movi    <#,0>,GOPRV
;
;Unwind stack to first procedure with line numbers.
;if unwound to monitor level, then not called inside procedure error

GO1:    tstb    <#,EV$MON>,<EVFLGS,4>,NOGO2
        jmpnzi  GOPRV,GO1A                       ;if called by non-priv, can't
        tstb    <#,EV$PRV>,<EVFLGS,4>,GO1B ;jump into priv. procedure
GO1A:   jmpnep  <PROC,4>,<#,NULL>,GO2   ;jump if named procedure
GO1B:   call    UNFRM                   ;else unwind
        jmp     GO1
;
; search thru proc. dir. for LINENO

GO2:    getval  <PROC,4>,r3             ;setup r3 as proc. dir.
        movi    <#,5>,r1                ;r1 is offset to line no.
        movi    NLINES,r2
GO3:    jmpzi   r2,NOGO3                ;jump if no line found
        subvec  r3,<#,TRANS>,r1         ;get proc. dir. lineno.
        jmpei   TRANS,LINENO,SETGO
        deci    r2
        inci    r1
        jmp     GO3
;
;line no. found, set up & go

SETGO:  call    UNSTK                   ;clear local stack
        movp    r4,<@,4>               ;reset local stack pointer
        movp    <#,TRANS>,r2            ;restore r2 as ind. code ptr.
        addi    NINTS,r1
        addi    NARGS,r1
        addi    NLOCS,r1
        subi    <#,4>,r1
        movp    r1,<LINE,4>
        jmp     SETLIN
```

```
;
;GO errors

NOGO1:  movi    <#,E$NALN>,ERRCOD       ;not a line number
        jmp     NOGO
NOGO3:  movi    <#,E$LNNF>,ERRCOD       ;line number not found

NOGO:   movp    <TOP,4>,r0
        movi    <#,2>,ERRCNT
        movp    <#,GOSTR>,r3
        movp    r0,r2
        movp    <#,1>,r1
        CALL    ERRARG,GIVEUP
        swapp   r0,ERRINF
        jmp     ERRHDL

NOGO2:  movi    <#,E$NIAP>,ERRCOD       ;not in a procedure
        movi    <#,0>,ERRCNT
        swapp   <#,EMPTY>,ERROBJ
        swapp   <#,EMPTY>,ERRINF        ;no context available
        jmp     ERRHDL

;[
;routine:DONE--returns from a procedure without value (UNDEF)
;attr: jump primitive
;]
        entry   DONE

DONE:   jmpnep  <#,NULL>,CURPRC,DONE1   ;a DONE in a RUN proc refers to the
                                        ; procedure containing the RUN
        call    UNFRM                   ;remove top stack frame,restore CURPRC
        jmp     DONE

DONE1:  call    UNFRM
;
;RDONE — entry point to an implicit DONE for RUN procedures
;       ( the unframing is done by EVEOL )

RDONE:  movp    <TOP,4>,r0
        swapp   <#,UNDEF>,<@,0>                 ;return no value
RDONE2: jmppi   LEVEL,GETRST            ;if stack not completely unwound
                                        ; go check if UNDEF can be returned
                                        ; here

;
;LOGO is finished?

LGODON: return
;[
;routine:RTURN (RETURN)--return top of local stack to previous frame
;attr: jump primitive
;]
        vsect
RETVAL: ptr     NULL            ;temp. storage for returned value

        psect
        entry   RTURN

RTURN:  tstb    <#,EV$VAL>,<EVFLGS,4>,CMDERR ;RETURN not allowed in VALUE
;
;VALUE will return here at EOS (implicit RETURN)

RTURN1: movp    <TOP,4>,r0
        cmovp   <@,0>,RETVAL            ;get top of stack(value to be
                                        ; returned)
;unwind until not RUN

RTURN2: tstb    <#,EV$RUN>,<EVFLGS,4>,RTURN3
        jmp     RTURN4
RTURN3: call    UNFRM
        jmp     RTURN2

RTURN4: call    UNFRM
        movp    <TOP,4>,r0
        swapp   RETVAL,<@,0>
        jmp     GETRST
;
; Character routines for lexical scanner
;[
;routine:LEXGET--get addr. of source string pointer from current proc. dir.
;output reg: r0
;type: called by LEXSRC
;global: SRCOFF (in EVAL)
;]

LEXGET: getval  CURPRC,r0               ; get proc. dir. ptr.
        subvec  r0,<#,TRANS>,SRCOFF     ; TRANS gets source code ptr
        movp    TRANS,r0
        movp    <#,NULL>,TRANS
        return

;[
;routine:MLXGET--get addr. of source string pointer from stack for RUN proc.
;output reg: r0
;global: r4 (FRAME)
;type: called by LEXSRC
;]
```

```
MLXGET: movp    <LINE,4>,r0
        return

;[
;routine:LXRCH--supplies a char. to scanner
;               translates untouchables (entry point from this is LEXCVT)
;output reg:r0
;type:called by scanner, passed in LEXRCH
;attr:SAFE
;]

LXRCH:  push    r1
        movp    <#,EVBBUF>,r1
        call    BINP                    ;get char. from source string
        pop     r1

        tstb    <#,B$DONE>,EVBFLG,LXFAIL ;fail signals EOL
;
;entry point for converting list characters to their print forms

        entry   LEXCVT                  ;called from NOTEXT in LGONAM

LEXCVT: jmpei   r0,<#,U$RBRK>,LXRB
        jmpei   r0,<#,U$LBRK>,LXLB
        jmpei   r0,<#,U$SPCS>,LXSP
        succeed

LXRB:   movi    <#,"]">,r0
        succeed

LXLB:   movi    <#,"[">,r0
        succeed

LXSP:   movi    <#," ">,r0
        succeed

LXFAIL: fail

        endmod EVALUATOR
```

4.2.5 LGOARI.LIL

```
        search  LIL

        module  LGOARI — LOGO arithmetic primitives

        include ERRORS
        include CONSTS
        include FLAGS

        extrn   ERRCOD,ERROBJ

        extrn   STDERR, GIVEUP

        vsect

ARGCNT: int 0
ARGPTR: ptr     NULL
RESULT: ptr     NULL                    ;holds partial results
NUMLEN: int 0
        psect

;[
;routine: PROD,SUM (both SUM & LONGSUM, PRODUCT & LONGPRODUCT)
;input regs: std. primitive
;output regs: r0—sum or product of arg. buffer
;attr: primitive
CALLS:
        extrn   ADDNUM,MULNUM,CRNUM
;]
        entry   SUM,PROD

SUM:    push    r3                      ;r3 will addr. of routine to do arithmetic
        movp    <#,ADDNUM>,r3
        jmp     PRDSUM

PROD:   push    r3
        movp    <#,MULNUM>,r3

PRDSUM: jmpzi   r1,RETZER               ;return zero if no args.
        jmpei   r1,<#,1>,RETARG         ;return arg. if only one arg.
        deci    r1
        movi    r1,ARGCNT
        cmovp   <@,2>,RESULT           ;r1 will hold result
        movp    r2,ARGPTR

PRSUM1: decp    ARGPTR
        movp    <#,RESULT>,r1
        movp    ARGPTR,r2
        call    <@,3>
        swapp   r0,RESULT
        deci    ARGCNT
        jmppi   ARGCNT,PRSUM1

        pop     r3
        swapp   RESULT,r0
        succeed
;
;return zero

RETZER: movi    <#,1>,r1
        call    CRNUM
        succeed
;
;return only arg. give

RETARG: swapp   <@,2>,r0
        succeed
;
;[
;routine: DIFF (DIFFERENCE)
;regs: std. normal primitive (r1 ignored)
;attr: primitive
;CALLS: ADDNUM
;]
        entry   DIFF

DIFF:   movp    r2,r1
        decp    r1                      ;r1 points to 2nd arg.
        gettyp  <@,1>,r0               ;get sign of 2nd arg.
        xori    <#,F$SIGN>,r0           ;complement sign (for subtraction)
        settyp  r0,<@,1>
        call    ADDNUM
        succeed

;[
;routine: DIVIDE
;regs: std. normal primitive (r1 ignored)
;attr: primitive
;CALLS:
        extrn   DIVNUM
;]
        entry   DIVIDE

DIVIDE: movp    r2,r1
        decp    r2
        call    DIVNUM,DIVERR
        swapp   <#,NULL>,r1             ;dealc. remainder
        succeed

;[
;routine: REM (REMAINDER)
;regs: std. normal primitive (rl ignored)
;attr: primitive
;CALLS: DIVNUM
;]
        entry   REM

REM:    movp    r2,r1
        decp    r2
        call    DIVNUM,DIVERR
        swapp   r1,r0                   ;return remainder (dealc. quot.)
        succeed
;
;divide by zero

DIVERR: movi    <#,E$DIV0>,ERRCOD
        jmp     STDERR
;
;[
;routine: MINUS (unary minus primitive)
;reg: std. normal primitive
;attr: primitive
;]
        entry   MINUS

MINUS:  swapp   <@,2>,r0                ;get arg.
        subvec  r0,<#,NUMLEN>           ;get length
        jmpei   NUMLEN,<#,1>,MINDON     ;if zero, return number

        gettyp  r0,r1
        xori    <#,F$SIGN>,r1           ;else flip sign
        settyp  r1,r0
MINDON: succeed

        endmod LGOARI
```

4.2.6 LGOBCO.LIL

```
        search  LIL
        module  LGOBOO -- LOGO primitives which yield boolean results

        include ERRORS
        include CONSTS
        include FLAGS

        extrn   CKBOOL,CMPMAG,CVTSTR,CVTNUM,GIVEUP
        extrn   TRUEP, FALSP
        extrn   STDERR, ERRCOD, ERROBJ
        extrn   EMPTY

;
;        Relational operators
;
; All the relational primitives work by setting
; 3 bits in r1 which describe whether to return "true or "false if
; their  1st operand is less than, equal to, or greater than the second,
;  and then calling RELCMP which does the actual comparison.
;  An additional two bits in r1 signify whether a case-sensitive
; comparison is to be performed and whether lists may be compared
; with each other (for EQUAL?, NOTEQUAL?, etc.).

;[
;routines: LESSP,GRTRP, EQUALP, GREP, LSEP, NOTEQP
;input regs: std. normal primitive
;output regs: r0 -returns "true or "false
;]
        entry LESSP,GRTRP, EQUALP, GREP, LSEP, NOTEQP

        const   LSTFLG,BIT$4            ; flag for list comparisons
        const   CASFLG,BIT$5            ; flag for case sensitivity

        const   LSSFLG,BIT$1

LESSP:  movi    <#,LSSFLG>,r1
        jmp     RELCMP

        const   GRTFLG,BIT$3

GRTRP:  movi    <#,GRTFLG>,r1
        jmp     RELCMP

        const   EQUFLG,BIT$2
        const   EQPFLG,EQUFLG+LSTFLG

EQUALP: movi    <#,EQPFLG>,r1
        jmp     RELCMP
        const   NOTFLG,BIT$1+BIT$3,+LSTFLG

NOTEQP: movi    <#,NOTFLG>,r1
        jmp     RELCMP

        const   LSEFLG,BIT$1+BIT$2

LSEP:   movi    <#,LSEFLG>,r1
        jmp     RELCMP

        const   GREFLG,BIT$2+BIT$3

GREP:   movi    <#,GREFLG>,r1
        jmp     RELCMP
;[
;routines: IDENT,PRCDEP,FOLLWP
;input regs: std. prim
;output regs: TRUEP or FALSP in r0
;  these routines all jump directly to STRCMP, since numbers are
;   to be treated as strings when passed to these operations.
;]

        entry   IDENT,PRCDEP,FOLLWP

        const   IDNFLG,EQPFLG+CASFLG

IDENT:  movi    <#,IDNFLG>,r1
        jmp     CASCMP

        const   PRCFLG,LSSFLG+CASFLG

PRCDEP: movi    <#,PRCFLG>,r1
        jmp     CASCMP

        const   FLWFLG,GRTFLG+CASFLG

FOLLWP: movi    <#,FLWFLG>,r1


CASCMP: decp    r2
        push    r3              ; set things up for STRCMP code
        movi    r1,r3
        jmp     STRCMP          ; and go there!

;
;do actual comparision

RELCMP: push    r3
```

```
        movi    r1,r3
        movi    <#,1>,r1
        call    CVTNUM,STCNV2           ;try to convert to number
                                        ;jump if not possible
;
;1st arg. is a number

        decp    r2                      ;point to 2nd arg.
        call    CVTNUM,STCNV1           ;try to convert 2nd arg.
                                        ;if not possible, convert 1st to string
;
;both args. are numbers

NUMCMP: gettyp  <@,2>,r0
        gettyp  <WORDSZ,2>,r1           ;get signs of numbers
        andi    <#,F$SIGN>,r0
        andi    <#,F$SIGN>,r1
        jmpei   r0,r1,NUMCM2            ;signs same, compare magnitudes
        jmpzi   r0,LSSCMP               ;jump if 2nd arg. positive
        jmp     GRTCMP

NUMCM2: movp    r2,r1
        jmpzi   r0,NUMCM3               ;jump if both are positive
        incp    r1
        jmp     NUMCM4
NUMCM3: incp    r2
NUMCM4: call    CMPMAG
        jmpzi   r0,EQUCMP
        jmppi   r0,LSSCMP
        jmp     GRTCMP
;
;1st arg. is number, 2nd is string

STCNV1: incp    r2                      ;r2 points back to 1st arg.
        call    CVTSTR,GIVEUP
        decp    r2
        jmp     STRCMP
;
;2nd arg. is of unknown type, 1st is string

STCNV2: decp    r2
        call    CVTSTR,GIVEUP
;
;string compare--can't compare lists

STRCMP: tstb    <#,LSTFLG>,r3,STRCML     ; can lists be compared???
        jmpep   <@,2>,<#,EMPTY>,STRCM2 ; no, error if any list args
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,RELER1
STRCM2: jmpep   <WORDSZ,2>,<#,EMPTY>,STRCM3
        gettyp  <WORDSZ,2>,r0
        tstb    <#,F$LIST>,r0,RELER2
        jmp     STRCM3                  ; go do comparison
STRCML: jmpep   <@,2>,<#,EMPTY>,STRCM3 ; EMPTY can compare w/ anything
        jmpep   <WORDSZ,2>,<#,EMPTY>,STRCM3
        gettyp  <@,2>,r0               ; make sure args are same type
        gettyp  <WORDSZ,2>,r1           ; as each other
        andi    <#,F$LIST>,r0
        andi    <#,F$LIST>,r1
        jmpei   r0,r1,STRCM3
        jmpzi   r0,RELER2
        jmp     RELER1
STRCM3: tstb    <#,CASFLG>,r3,STRCM4    ; ignore lowercase chars?
        cmpsu   <@,2>,<WORDSZ,2>,LSSCMP,GRTCMP          ; yes.
        jmp     EQUCMP
STRCM4: cmps    <@,2>,<WORDSZ,2>,LSSCMP,GRTCMP          ; no.
;
;strings or numbers are equal

EQUCMP: tstb    <#,EQUFLG>,r3,RETTRU
        jmp     RETFLS
;
;1st string or number is greater than 2nd

GRTCMP: tstb    <#,GRTFLG>,r3,RETTRU
        jmp     RETFLS
;
;2nd string or number is greater than 1st

LSSCMP: tstb    <#,LSSFLG>,r3,RETTRU

RETFLS: movp    <#,FALSP>,r0
        pop     r3
        succeed

RETTRU: movp    <#,TRUEP>,r0
        pop     r3
        succeed

RELER2: incp    r2
RELER1: movi    <#,E$WELP>,ERRCOD
        pop     r3
        jmp     STDERR
;
;[
;routine: ANY ALL -- does computation for ANY?,ALL?,EITHER?,BOTH?
```

388

```
;regs: std. prim.
;calls: CKBOOL
;attr:SAFE
;]
        vsect

CHECKB: ptr     NULL                        ;the boolean on which ANYALL ends
                                            ;(FALSP for ALL?, TRUEP for ANY?)
NCHECK: ptr     NULL                        ;not CHECKB
RESULT: ptr     NULL                        ;of ANYALL

        psect

        entry   ANY,ALL

ANY:    movp    <#,TRUEP>,CHECKB
        movp    <#,FALSP>,NCHECK
        jmp     ANYALL

ALL:    movp    <#,FALSP>,CHECKB
        movp    <#,TRUEP>,NCHECK
;       jmp     ANYALL

ANYALL: movp    NCHECK,RESULT               ;default result
ANYAL1: jmpzi   r1,ANYDON                   ;return RESULT if arg. list exhausted
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,BOOERR        ;boolean are always words
        jmpnep  <@,2>,CHECKB,ANYAL2
;
;found desired boolean--set RESULT

        movp    <@,2>,RESULT
        jmp     ANYAL3

ANYAL2: jmpep   <@,2>,NCHECKB,ANYAL3       ;found undesired boolean--keep going
;
; found word of unknown type--call CKBOOL

        pushi   r1
        movp    <#,NULL>,r1
        movs    <@,2>,r1                   ;copy arg.
        call    CKBOOL,BOOERR               ;error if non-boolean string
        popi    r1
        jmpnep  r0,CHECKB,ANYAL3
;
;found desired boolean--set RESULT

        movp    r0,RESULT
```

```
ANYAL3: deci    r1
        decp    r2
        jmp     ANYAL1

ANYDON: movp    RESULT,r0
        succeed
;
;[
;routine: NOT
;regs: std. normal primitive
;attr: UNSAFE
;calls: CKBOOL
;]
        entry   NOT

NOT:    gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,BOOERR
        jmpep   <@,2>,<#,TRUEP>,RTNFLS
        jmpep   <@,2>,<#,FALSP>,RTNTRU
        cmovp   <@,2>,r1
        call    CKBOOL,NOTERR
        jmpep   r0,<#,TRUEP>,RTNFLS
;
; Return TRUE or FALSE for most LGOBOO primitives

RTNTRU: movp    <#,TRUEP>,r0
        succeed

RTNFLS: movp    <#,FALSP>,r0
        succeed

NOTERR: movi    <#,E$NBIC>,ERRCOD
        swapp   r1,<@,2>               ;put arg. back on stack for PFAIL
        jmp     STDERR
;
; non-boolean passed as arg.

BOOERR: swapp   <#,NULL>,r1            ;deallocate copy of arg.
        popi    r1
        movi    <#,E$NBIC>,ERRCOD
        jmp     STDERR

;
;[
;routines: WORDP (WORD?, returns "TRUE if arg. is a word)
;          LISTP (LIST?, semanitically equivalent to NOT WORD?)
;regs: std. normal primitives
```

```
;attr: SAFE
;]
        entry   WORDP,LISTP

LISTP:  gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,RTNTRU
        jmp     RTNFLS

WORDP:  gettyp  <@,2>,r0
        tstb    <#,F$NUM>,r0,RTNTRU     ;numbers are words
        jmpep   <#,EMPTY>,<@,2>,RTNTRU ;EMPTY is a word OR a list
        tstb    <#,F$LIST>,r0,RTNFLS
        jmp     RTNTRU

;[
;routine: INTP — INTEGER?
;regs: std. normal prim.
;attr:UNSAFE
;CALLS:
        extrn CVISN
;]
        entry   INTP

INTP:   gettyp  <@,2>,r1
        tstb    <#,F$NUM>,r1,RTNTRU     ;return true
        swapp   <@,2>,r1
        call    CVISN,INTFLS
INTTRU: swapp   <#,TRUEP>,r0            ;dealc number
        jmp     INTRET

INTFLS: swapp   <#,FALSP>,r0            ;dealc number

INTRET: swapp   <#,NULL>,r1             ;dealc input string
        succeed

;[
;ROUTINE:EMPTYP --EMPTY? predicate
;REGS:std. normal primitive
;ATTR:SAFE
;]
        entry   EMPTYP

EMPTYP: jmpep   <@,2>,<#,EMPTY>,RTNTRU
        jmp     RTNFLS

;[
;ROUTINE: ZEROP--ZERO? predicate
; REGS: std. normal prim.
```

```
;ATTR:UNSAFE
;CALLS:CVTNUM
;]
        vsect

NUMLEN: int 0

        psect

        entry   ZEROP

ZEROP:  movi    <#,1>,r1
        call    CVTNUM,RTNFLS
        subvec  <@,2>,<#,NUMLEN>
        jmpei   <#,1>,NUMLEN,RTNTRU
        jmp     RTNFLS
;
;WORKSPACE BOOLEANS
;
;[
;ROUTINE:THINGP (THING?)
;REGS: std. boolean primitive
;ATTR: SAFE
;GLOBALS:
        extrn NT
;]
        entry   THINGP

THINGP: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,RTNFLS
        tstb    <#,F$NUM>,r0,RTNFLS
        find    NT,<@,2>,r0
        jmpep   <#,NULL>,r0,RTNFLS      ;table entry for name doesn't exist
        getval  r0,r0
        jmpep   r0,<#,UNDEF>,RTNFLS     ;value is UNDEFined
        jmp     RTNTRU

;[
;ROUTINES:PROCP (PROCEDURE?) & PRIMP (PRIMITIVE?)
;REG:std. boolean primitive
;ATTR:SAFE
;GLOBAL:
        extrn PT, PRVUSR
;]

        entry   PROCP,PRIMP
;
;non-zero r1 means we are looking for primitives or privileged procs only
```

```
PROCP:  movi    <#,0>,r1
        jmp     PROCHK

PRIMP:  movi    <#,1>,r1

PROCHK: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,RTNFLS
        tstb    <#,F$NUM>,r0,RTNFLS
        find    PT,<@,2>,r0
        jmpep   <#,NULL>,r0,RTNFLS
        getval  r0,r2
        jmpep   <#,UNDEF>,r2,RTNFLS     ; it doesn't exist, return FALSE
        jmpzi   r1,RTNTRU               ;it is a procedure
        getflg  r0,r0
        tstb    <#,F$PRIM>,r0,RTNTRU    ;it is a primitive
        jmpnzi  PRVUSR,RTNFLS           ;LIL prims only for privileged user
        tstb    <#,F$PRIV>,r0,RTNTRU
        jmp     RTNFLS

;[
;ROUTINE: TEXTP (TEXT?)
;REGS: std. normal prim
;GLOBALS: PT
;ATTR: SAFE
;]
        entry   TEXTP

TEXTP:  gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,RTNFLS
        tstb    <#,F$NUM>,r0,RTNFLS
        find    PT,<@,2>,r0
        jmpep   r0,<#,NULL>,RTNFLS
        getval  r0,r1
        jmpep   r1,<#,UNDEF>,RTNFLS
        getflg  r0,r1
        tstb    <#,F$PRIM>,r1,RTNFLS
        getval  r0,r1                   ;get proc. dir.
        jmpep   r1,<#,UNDEF>,RTNFLS     ;proc. has been erased
        gettyp  r1,r0
        andi    <#,F$TEXT>,r0
        jmpnzi  r0,RTNFLS
        jmp     RTNTRU
;
;[
;ROUTINE:FLATP (FLATLIST?—returns TRUE if list with no sub-lists)
;REGS:std. normal prim
;ATTR:SAFE
;]
        vsect

CHR:    int     0                       ;holds character from input object

        entry   FLATP

FLATP:  gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,FLATP2
        jmp     RTNFLS

FLATP2: getlen  <@,2>,r0               ;r0 is index into string
FLATP3: jmpzi   r0,RTNTRU               ;contents of list checked,no sub-lists
        substr  <@,2>,<#,CHR>,r0
        jmpei   CHR,<#,U$RBRK>,RTNFLS   ;search from back, find internal
        deci    r0                      ; right bracket first
        jmp     FLATP3
        endmod  LGOBOO
```

4.2.7 LGODAT.LIL

```
        search  LIL

        module LGODAT —LOGO variables & data tables

        include CONSTS
        include FLAGS
        include ERRORS

        vsect                                ;write-enabled area

; Hash tables

        pfix    F$VEC, RWTSIZ
RWTTAB: int     RWTSIZ
        blkp    RWTSIZ-1                     ;reserved word table

        pfix    F$VEC, PTSIZ
PTTAB:  int     PTSIZ
        blkp    PTSIZ-1         ;procedure table

        pfix    F$VEC, NTSIZ
NTTAB:  int     NTSIZ
        blkp    NTSIZ-1         ;name table

        pfix    F$VEC,FTSIZ
FTTAB:  int     FTSIZ
        blkp    FTSIZ-1         ;file names

        pfix    F$VEC,SPTSIZ
SPTTAB: int     SPTSIZ
        blkp    SPTSIZ-1                     ;special literals
;
; bootstrap file table entry

        entry   BTSENT

BTSENT: ptr     NULL

        psect                                ;read-only data

        entry   EMPTY

        pfix    F$LIST+F$STR,0
EMPTY:
;
; symbol table names

        entry   RWT, PT, NT, FT, SPT
```

396

```
RWT:    ptr     RWTTAB
PT:     ptr     PTTAB
NT:     ptr     NTTAB
FT:     ptr     FTTAB
SPT:    ptr     SPTTAB

;
; The print names of the reserved symbols.
; These are used in both the scanner & evaluator

        entry   IF, ELSE, DO, THEN, REPT, WHIL, UNTL
        entry   BCMS, LESSOP, GRTROP, EQALOP
        entry   SUMOP, DIFFOP, PRODOP, QUOTOP, REMOP

        pfix    F$STR, 2
IF:     text    "if"
        pfix    F$STR, 4
ELSE:   text    "else"
        pfix    F$STR, 2
DO:     text    "do"
        pfix    F$STR, 4
THEN:   text    "then"
        pfix    F$STR, 6
REPT:   text    "repeat"
        pfix    F$STR, 5
WHIl:   text    "while"
        pfix    F$STR, 5
UNTL:   text    "until"
        pfix    F$STR, 2
BCMS:   byte(9) C$LANG,"-"              ; non MACRO-10 "<-"
        pfix    F$STR, 1
LESSOP: byte(9) C$LANG                  ; non MACRO-10 "<"
        pfix    F$STR, 1
GRTROP: byte(9) C$RANG                  ; non MACRO-10 ">"
        pfix    F$STR, 1
EQALOP: text    "="
        pfix    F$STR, 1
SUMOP:  text    "+"
        pfix    F$STR, 1
DIFFOP: text    "-"
        pfix    F$STR, 1
PRODOP: text    "*"
        pfix    F$STR, 1
QUOTOP: text    "/"
        pfix    F$STR, 1
REMOP:  text    "\"
```

```
;
; the print names of the primitives

        entry MAKSTR,DIFSTR,PRDSTR,LNGPRD,REMSTR,QUOSTR,MINSTR,LESSTR,GRTSTR
        entry EQUSTR,GRESTR,LSESTR,SUMSTR,LNGSUM,IDNSTR,PCDSTR,FLWSTR
        entry RUNSTR,VALSTR,TRTSTR,PVRSTR
        entry STPSTR,GOSTR,DONSTR,RETSTR,THGSTR
        entry TXFSTR,TEXSTR,NTXSTR,TXPSTR,PRCSTR,NAMSTR,IPRSTR,RPRSTR,INMSTR,PNMSTR
        entry STPSTR,USPSTR,STNSTR,USNSTR,PPPSTR,PPNSTR
        entry TOSTR,TOGSTR,TOQSTR,ERNSTR,ERPSTR
        entry FIRSTR,BTFSTR,SENSTR,LSNSTR,LSTSTR,LSNLST,FPTSTR,WRDSTR,LWRSTR,ASCSTR,CHRS
        entry CNTSTR,ELMSTR,SBJSTR,MEMSTR
        entry WRPSTR,INPSTR,INSSTR,ZERSTR,EPPSTR
        entry THPSTR,PMPSTR,PRPSTR,LSPSTR,FLPSTR
        entry LASSTR,BTLSTR,LPTSTR
        entry PRTSTR,LPRSTR,OPNSTR,RDCSTR,TYPSTR,LTYSTR
        entry FPRSTR,FWTSTR,WTCSTR,FTYSTR
        entry WTLSTR,EFLSTR,FLSSTR,ACTSTR,CPNSTR,SCPSTR,COLSTR,RDLSTR
        entry CLSSTR,UOPSTR,DELSTR,RFNSTR,IPCSTR
        entry PRFSTR,PSFSTR
        entry ALLSTR,EITSTR,ANYSTR,BOTSTR,NOTSTR
        entry PPTSTR,TPSSTR,TPHSTR,SPTSTR,BRTSTR
        entry STTSTR,STMSTR,UNWSTR,UWWSTR,LEVSTR
        entry   DATSTR,TMESTR,VRSSTR,WAISTR
        entry   TIMSTR,STISTR,CTISTR

        pfix    F$STR,5
LEVSTR: text "LEVEL"

        pfix    F$STR,7
STMSTR: text "#SETMON"

        pfix    F$STR,4
DATSTR: text "DATE"

        pfix    F$STR,4
TMESTR: text "TIME"

        pfix    F$STR,7
VRSSTR: text "VERSION"

        pfix    F$STR,4
WAISTR: text "WAIT"

        pfix    F$STR,6
UNWSTR: text "UNWIND"

        pfix    F$STR,10
```

```
UWWSTR: text "UNWINDWITH"

        pfix    F$STR,7
STTSTR: text "SETTRAP"

        pfix    F$STR,7
PPTSTR: text "POPTRAP"

        pfix    F$STR,8
TPSSTR: text "TRAPSET?"

        pfix    F$STR,10
SPTSTR: text "SPRINGTRAP"

        pfix    F$STR,11
TPHSTR: text "TRAPHANDLER"

        pfix    F$STR,9
BKTSTR: text "BACKTRACE"

        pfix    F$STR,8
STISTR: text "SETTIMER"

        pfix    F$STR,10
CTISTR: text "CLEARTIMER"

        pfix    F$STR,5
TIMSTR: text "TIMER"

        pfix    F$STR,5
WRPSTR: text "WORD?"

        pfix    F$STR,6
THPSTR: text "THING?"

        pfix    F$STR,10
PMPSTR: text "PRIMITIVE?"

        pfix    F$STR,10
PRPSTR: text "PROCEDURE?"

        pfix    F$STR,5
LSPSTR: text "LIST?"

        pfix    F$STR,9
FLPSTR: text "FLATLIST?"

        pfix    F$STR,5
```

```
ZERSTR: text "ZERO?"

        pfix    F$STR,6
EPPSTR: text "EMPTY?"

        pfix    F$STR,8
INPSTR: text "INTEGER?"

        pfix    f$STR,11
INSSTR: text "INSERT$LINE"

        pfix    F$STR,4
ALLSTR: text "ALL?"

        pfix    F$STR,5
BOTSTR: text "BOTH?"

        pfix    F$STR,3
NOTSTR: text "NOT"

        pfix    F$STR,4
ANYSTR: text "ANY?"

        pfix    F$STR,7
EITSTR: text "EITHER?"

        pfix    F$STR,4
LASSTR: text "LAST"

        pfix    F$STR,7
BTLSTR: text "BUTLAST"

        pfix    F$STR,7
LPTSTR: text "LASTPUT"

        pfix    F$STR,5
ASCSTR: text "ASCII"

        pfix    F$STR,4
CHRSTR: text "CHAR"

        pfix    F$STR,5
CNTSTR: text "COUNT"

        pfix    F$STR,7
ELMSTR: text "ELEMENT"

        pfix    F$STR,9
```

```
SBJSTR: text "SUBOBJECT"

        pfix    F$STR,6
MEMSTR: text "MEMBER"

        pfix    F$STR,4
TYPSTR: text "TYPE"

        pfix    F$STR,8
LTYSTR: text "LONGTYPE"

        pfix    F$STR,4
OPNSTR: text "OPEN"

        pfix    F$STR,5
CLSSTR: text "CLOSE"


        pfix    F$STR,6
DELSTR: text "DELETE"

        pfix    F$STR,6
RENSTR: text    "RENAME"

        pfix    F$STR,6
UOPSTR: text "UNOPEN"

        pfix    F$STR,8
RDLSTR: text "READLINE"

        pfix    F$STR,8
RDCSTR: text     "READCHAR"

; READ is defined in special words

        pfix F$STR,5
PRTSTR: text "PRINT"

        pfix F$STR,9
LPRSTR: text "LONGPRINT"

        pfix    F$STR,9
FPRSTR: text "FULLPRINT"

        pfix    F$STR,9
FWTSTR: text "FULLWRITE"
```

```
        pfix    F$STR,9
WTCSTR: text "WRITECHAR"

        pfix    F$STR,9
WTLSTR: text "WRITELINE"

        pfix    F$STR,8
EFLSTR: text "ENDFILE?"

        pfix    F$STR,6
IPCSTR: text    "INPUT?"

        pfix    F$STR,8
FLSSTR: text "FILESIZE"

        pfix    F$STR,10
ACTSTR: text "ACCESSTYPE"

        pfix    F$STR,11
CPNSTR: text "CHARPOINTER"

        pfix    F$STR,14
SCPSTR: text "SETCHARPOINTER"

        pfix    F$STR,6
COLSTR: text "COLUMN"

        pfix    F$STR,8
FTYSTR: text "FULLTYPE"
        pfix    F$STR,10
PRFSTR: text "$PRINTFORM"

        pfix    F$STR,10
PSFSTR: text "$PARSEFORM"

        pfix F$STR,5
FIRSTR: text "FIRST"

        pfix F$STR,8
BTFSTR: text "BUTFIRST"

        pfix F$STR,8
SENSTR: text "SENTENCE"

        pfix F$STR,12
LSNSTR: text "LONGSENTENCE"

        pfix F$STR,8
```

```
LSNLST: text "LONGLIST"

        pfix F$STR,4
LSTSTR: text "LIST"

        pfix F$STR,8
FPTSTR: text "FIRSTPUT"

        pfix F$STR,4
WRDSTR: text "WORD"

        pfix F$STR,8
LWRSTR: text "LONGWORD"

        pfix    F$STR,2
GOSTR:  text "GO"

        pfix F$STR,4
DONSTR: text "DONE"

        pfix F$STR,6
RETSTR: text "RETURN"

        pfix    F$STR,3
RUNSTR: text "RUN"

        pfix    F$STR,8
PVRSTR: text "$PRIVRUN"

        pfix    F$STR,5
VALSTR: text "VALUE"

        pfix    F$STR,10
TRTSTR: text "TRAPRETURN"

        pfix    F$STR,4
MAKSTR: text "MAKE"

        pfix    F$STR,5
THGSTR: text    "THING"

        pfix    F$STR,10
DIFSTR: text "DIFFERENCE"

        pfix    F$STR,3
SUMSTR: text "SUM"

        pfix    F$STR,7
```

```
LNGSUM: text "LONGSUM"

        pfix    F$STR,7
PRDSTR: text "PRODUCT"

        pfix    F$STR,11
LNGPRD: text "LONGPRODUCT"

        pfix    F$STR,9
REMSTR: text "REMAINDER"

        pfix    F$STR,8
QUOSTR: text "QUOTIENT"

        pfix    F$STR,5
MINSTR: text "MINUS"

        pfix    F$STR,5
LESSTR: text "LESS?"

        pfix    F$STR,8
GRTSTR: text "GREATER?"

        pfix    F$STR,8
FLWSTR: text "FOLLOWS?"

        pfix    F$STR,9
PCDSTR: text "PRECEDES?"

        pfix    F$STR,11
EQUSTR: text "EQUIVALENT?"

        pfix    F$STR,10
IDNSTR: text "IDENTICAL?"

        pfix    F$STR,10
GRESTR: text "GREATEREQ?"

        pfix    F$STR,7
LSESTR: text "LESSEQ?"

        pfix    F$STR,6
STFSTR: text    "$STUFF"

        pfix    F$STR,4
TEXSTR: text "TEXT"

        pfix    F$STR,9
```

```
TXFSTR: text "$TEXTFORM"

        pfix    F$STR,5
TXPSTR: text "TEXT?"

        pfix    F$STR,6
NTXSTR: text "NOTEXT"

        pfix    F$STR,21
STPSTR: text "SETPROPERTY$PROCEDURE"

        pfix    F$STR,23
USPSTR: text "UNSETPROPERTY$PROCEDURE"

        pfix    F$STR,16
STNSTR: text "SETPROPERTY$NAME"

        pfix    F$STR,18
USNSTR: text "UNSETPROPERTY$NAME"

        pfix    F$STR,14
PPNSTR: text "PROPERTY?$NAME"

        pfix    F$STR,19
PPPSTR: text "PROPERTY?$PROCEDURE"

        pfix    F$STR,9
PRCSTR: text "USERPROCS"

        pfix    F$STR,9
NAMSTR: text "USERNAMES"

        pfix    F$STR,9
IPRSTR: text "INV$PROCS"

        pfix    F$STR,9
RPRSTR: text "PRV$PROCS"

        pfix    F$STR,9
INMSTR: text "INV$NAMES"

        pfix    F$STR,9
PNMSTR: text "PRV$NAMES"

        pfix    F$STR,2
TOSTR:  text "TO"

        pfix    F$STR,8
```

```
TOGSTR: text "TOGREEDY"

        pfix    F$STR,9
TOQSTR: text "TOQUOTING"

        pfix    F$STR,15
ERPSTR: text "ERASE#PROCEDURE"

        pfix    F$STR,10
ERNSTR: text "ERASE#NAME"

;
; the print names of the read-only names

        entry   EMPSTR,BOOSTR
        entry   TPRSTR,TCDSTR,TLNSTR,TOBSTR,TCNSTR,TLVSTR

        pfix    F$STR,13
TPRSTR: text "TRAPPROCEDURE"

        pfix    F$STR,8
TCDSTR: text "TRAPCODE"

        pfix    F$STR,8
TLNSTR: text "TRAPLINE"

        pfix    F$STR,9
TLVSTR: text "TRAPLEVEL"

        pfix    F$STR,10
TOBSTR: text "TRAPOBJECT"

        pfix    F$STR,11
TCNSTR: text "TRAPCONTEXT"

        pfix    F$STR,5
EMPSTR: text "EMPTY"

        pfix    F$STR,6
BOOSTR: text "#BOOTS"                  ;name which holds name of bootstrap
                                       ;file

;
; standard file names

        entry   CSLSTR,BTSSTR

        pfix    F$STR,7
```

406

```
CSLSTR: text "CONSOLE"          ;default user file

        pfix    F$STR,10
BTSSTR: text    "BOOTS.LGO "            ;LOGO bootstrap file

;
; special literals recognized by LOGO primitives

        entry   ACSSTR,ERRSTR,CPTSTR,EOFSTR
        entry   WRTSTR,RDSTR,RNDSTR,APPSTR, INTSTR,UPNSTR
        entry   PARSTR,EVLSTR,FILSTR,ALCSTR,TRCSTR,CLKSTR,KBDSTR,PANSTR
        entry   QUTSTR,NORSTR,GRDSTR,EWOSTR
        entry   PPRSTR,PINSTR,PTRSTR,PSTSTR,PNRSTR,PNWSTR

        pfix    F$STR,9
PPRSTR: text "PRIMITIVE"

        pfix    F$STR,9
PINSTR: text "INVISIBLE"

        pfix    F$STR,6
PTRSTR: text "TRACED"

        pfix    F$STR,7
PSTSTR: text "STEPPED"

        pfix    F$STR,6
PNRSTR: text "NOREAD"

        pfix    F$STR,7
PNWSTR: text "NOWRITE"

        pfix    F$STR,7
QUTSTR: text "QUOTING"

        pfix    F$STR,6
NORSTR: text "NORMAL"

        pfix    F$STR,6
GRDSTR: text "GREEDY"

        pfix    F$STR,4
EWOSTR: text "EWOC"

        pfix    F$STR,5
PARSTR: text "PARSE"

        pfix    F$STR,9
```

```
EVLSTR: text "EVALUATOR"

        pfix    F$STR,4
FILSTR: text    "FILE"

        pfix    F$STR,5
ALCSTR: text "STORE"

        pfix    F$STR,5
TRCSTR: text "TRACE"

        pfix    F$STR,5
CLKSTR: text "CLOCK"

        pfix    F$STR,8
KBDSTR: text "KEYBOARD"

        pfix    F$STR,5
PANSTR: text "PANIC"

        pfix    F$STR,6
ACSSTR: text "ACCESS"

        pfix    F$STR,5
ERRSTR: text    "ERROR"

        pfix    F$STR,7
CPTSTR: text "CHARPTR"

        pfix    F$STR,3
EOFSTR: text "EOF"

        pfix    F$STR,11
INTSTR: text "INTERACTIVE"
        pfix    F$STR,8
UPNSTR: text "UNOPENED"


        pfix    F$STR,5
WRISTR: text "WRITE"

        pfix    F$STR,4
RDSTR:  text "READ"

        pfix    F$STR,6
APPSTR: text "APPEND"

        pfix    F$STR,6
```

```
RNDSTR: text "RANDOM"

        endmod  LGODAT
```

4.2.8 LGOGAR.LIL

```
        search  LIL

        module  LGOGAR — LOGO garbage collector and erase primitives

        include FLAGS
        include CONSTS
        include ERRORS

        extrn   NT,PT
        extrn   GCNAME,TRPPRC
        extrn   STDERR,ERRCOD,CMDDON

        const   F$PVTR,F$PRIV+F$TRCE    ;privileged ored with trace flag
;
;  variables:

        vsect

PARVEC: ptr     null            ; pointer to parse vector in LGOGAR
TMPPTR: ptr     null
HSHLST: ptr     null            ; pointer to current entry in hash list

NINTS:  int     0               ; buffer for first 4 words of proc. dir.
NLOCS:  int     0
NARGS:  int     0
NLINES: int     0

PVINDX: int     0               ; index in parse vector
SYMIND: int     0               ; current offset in symbol (hash) table
SYMLEN: int     0               ; length of symbol table

        psect

OMMARK:                 ; table for mark operation to sort out what to
                        ; do with a given OM opcode
        int     2               ; litc
        int     -1              ; valc
        int     1              ; dref
        int     2              ; jump
        int     2              ; jmpi
        int     2              ; jmpw
        int     2              ; jmpu
        int     2              ; decj
        int     -2             ; ncal
        int     0              ; eol
        int     1              ; eos

        const   MRKJMP, OMMARK-OMFAKE
;[
;routine: GARBAG
;       expunges all undefined entries in the procedure and name
;       tables that are not referenced in any parsed procedure lines.
;input, output regs: none (preserves all regs)
;calls: from the evaluator, whenever top level is reached and an
;       ERASE has occurred.
;attr: SAFE
;]

        entry   GARBAG

GARBAG: pushm   r0,r3
        movi    <#,2>,SYMIND            ; initialize proc. table index
        subvec  PT,<#,SYMLEN>,<#,1>     ; get length of table

PTLOOP: subvec  PT,<#,HSHLST>,SYMIND
                               ; get head of list for one hash table entry
                               ; (jump to PTDONE if hash table exhausted)
        jmpep   HSHLST,<#,NULL>,NEXTPT  ; if list empty, advance to next
                                        ; entry in PT
PRLOOP: getval  HSHLST,r1               ; get proc. dir. addr in r1
        jmpep   r1,<#,UNDEF>,NEXTPR     ; skip it if proc is undefined
        getflg  HSHLST,r2               ; if defined, get flags
        tstb    <#,F$PRIM>,r2,NEXTPR  ; don't scan parse if primitive
        subvec  r1,<#,NINTS>,<#,1>,<#,4>     ; otherwise get proc dir
        movi    NARGS,r2                ; calculate # of locals+args
        addi    NLOCS,r2
        movi    NINTS,r0                ; r0 points just before 1st local

MRKLOC: jmpzi   r2,MRKLDON     ; loop to mark local and argument names
        inci    r0
        subvec  r1,<#,TMPPTR>,r0        ; get addr. of entry for local
        getflg  TMPPTR,r3               ; and mark each one
        setb    <#,F$MARK>,r3
        setflg  r3,TMPPTR
        deci    r2
        jmp     MRKLOC

MKLDON: movi    r0,PVINDX                       ; save offset in proc. dir.

MRKREF: jmpzi   NLINES,NEXTPR                   ; loop thru line parse vectors
        inci    PVINDX
        subvec  r1,<#,PARVEC>,PVINDX            ; get address of line's parsed code
        jmpep   <#,NULL>,PARVEC,PVDONE          ; dont look at NULL parse vector
        movi    <#,2>,r0                        ; offset of first instr. in parse

PVLOOP: subvec  PARVEC,<#,TMPPTR>,r0            ; loop thru OM code in a parse vec.
        movp    TMPPTR,r3                       ; get OM opcode
        movi    <MRKJMP,3>,r3                   ; get mark function code
        jmpzi   r3,PVDONE                       ; if 0, EOL was reached
        jmppi   r3,PVNEXT                       ; if +, not an NCAL or VALC
        inci    r0                              ; if -, mark follwing item...
        subvec  PARVEC,<#,TMPPTR>,r0            ;... in parse vec (proc or name)
        getflg  TMPPTR,r2
        setb    <#,F$MARK>,r2
        setflg  r2,TMPPTR
        subi    r3,r0           ; (r3 is neg. of offset to next instr
        jmp     PVLOOP          ;  in parse vector)

PVNEXT: addi    r3,r0           ; nothing to mark, r3 is offset to next
        jmp     PVLOOP          ; OM instruction in parse vector.

PVDONE: deci    NLINES
        jmp     MRKREF

NEXTPR: getnxt  HSHLST                          ; get next item in proc. dir.
        jmpnep  <#,NULL>,HSHLST,PRLOOP          ; and loop if hash list not finished

NEXTPT: inci    SYMIND                          ; advance to next PT hash entry
        jmpngi  SYMIND,SYMLEN,PTLOOP

PTDONE: movp    PT,r1
        call    SYMREL                          ; mark phase completed,
        movp    NT,r1                           ; expunge garbage from NT & PT
        call    SYMREL
        popm    r0,r3
        return

;
;[
;routine: SYMREL — releases unreferanced, undefined symbols from a
;         symbol table and clears the mark bits of all remaining symbols.
;input regs: r1—address of name table to be cleaned up
;output regs: none
;               (r1-r3 destroyed)
;attr: SAFE
;]

SYMREL: movi    <#,2>,SYMIND            ; init. offset to first hash entry
        subvec  r1,<#,SYMLEN>,<#,1>

SYGCLP: subvec  r1,<#,HSHLST>,SYMIND
        jmpep   HSHLST,<#,NULL>,GCNXSY
HLGCLP: movp    HSHLST,TMPPTR
        getnxt  HSHLST                  ; remember addr. of next entry
        getflg  TMPPTR,r2
        tstb    <#,F$MARK>,r2,GCNXHL  ; skip if marked or priv
        tstb    <#,F$PRIV>,r2,GCNXHL
        getval  TMPPTR,r0               ; look at value
        jmpnep  r0,<#,UNDEF>,GCNXHL     ; skip if defined
        getnam  TMPPTR,r3
        remove  r1,r3                   ; deallocate nametable entry
        jmp     GCNXH1

GCNXHL: clrb    <#,F$MARK>,r2           ; clear mark flag
        setflg  r2,TMPPTR               ; on last object checked
GCNXH1: jmpnep  HSHLST,<#,NULL>,HLGCLP  ; loop if hash list not exhausted

GCNXSY: inci    SYMIND
        jmpngi  SYMIND,SYMLEN,SYGCLP

SYDONE: return

;
;[
;routine: ERANAM (ERASE#NAME) — erases a name
;input regs: std. prim.
;       takes 1 arg, which is name of symbol to be erased
;output regs:  UNDEF in r0
;attr: SAFE
;]

        entry   ERANAM

ERANAM: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,ERAWLF     ; symbol name must be word
        find    NT,<@,2>,r1             ; look up name in name table
        jmpep   <#,NULL>,r1,ERANXO       ; error if not found
        getval  r1,r0
        jmpep   <#,UNDEF>,r0,ERAUDF      ; error if undefined
        getflg  r1,r0
        tstb    <#,EV$PRV>,<EVFLGS,4>,ERANM1  ; if running non-priv.
        tstb    <#,F$PRIV>,r0,ERAOPV    ; check for privileged name
ERANM1: clrb    <#,F$PVTR>,r0           ; clear any trace/priv flags
        setflg  r0,r1
        setval  <#,UNDEF>,r1            ; and formally undefine name
        movi    <#,1>,GCNAME            ; signal that garbage is present
        jmp     CMDDON

;[
;routine: ERAPRC (ERASE#PROCEDURE) -- erases a procedure
;input, output: as ERANAM
;]

        entry   ERAPRC

ERAPRC: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,ERAWLF    ; symbol name must be word
        find    PT,<@,2>,r1            ; look up proc in proc table
        jmpep   <#,NULL>,r1,ERANXO      ; error if not found
        getval  r1,r0
        jmpep   <#,UNDEF>,r0,ERAUDF     ; error if undefined
        getflg  r1,r0
        tstb    <#,EV$PRV>,<EVFLGS,4>,ERAPR1  ; if running non-priv.
        tstb    <#,F$PRIV>,r0,ERAOPV    ; check for privileged name
ERAPR1: tstb    <#,F$PRIM>,r0,ERAOPV    ; can't erase a primitive!
        pushm   r2,r3
        movi    <#,NTRAPS>,r2
        cvtip   r2,r3           ; check that procedure isn't in the
        decp    r3              ; trap tables as a handler...
ERAPR2: movp    <TRPPRC,3>,r0           ; get pointer to top trap data
ERAPR3: jmpep   <#,NULL>,r0,ERAPR4      ;..block, and check the list
        subvec  r0,<#,TMPPTR>,<#,3>
        jmpep   r1,TMPPTR,ERAPTT        ; error if entry addr found
        subvec  r0,<#,TMPPTR>,<#,4>     ; else travel down link ptr
        movp    TMPPTR,r0
        jmp     ERAPR3
ERAPR4: decp    r3              ; decrement count of trap types
        deci    r2
        jmppi   r2,ERAPR2
        popm    r2,r3
        movp    r4,r0           ; check that procedure isn't active
ERAPR5: jmpep   <#,NULL>,<PROC,0>,ERAPR6  ; skip run frame
        jmpep   <PROC,0>,r1,ERAPAC       ; error if proc. is active
ERAPR6: movp    <PREV,0>,r0             ; loop on next frame
        jmpnep  <#,STACK>,r0,ERAPR5
        getflg  r1,r0
        clrb    <#,F$PVTR>,r0           ; clear any trace/priv flags
        setflg  r0,r1
        setval  <#,UNDEF>,r1
        movi    <$,1>,GCNAME            ; signal that garbage is present
        jmp     CMDDON

; error conditions:

ERAWLF: movi    <$,E$WELF>,ERRCOD
        jmp     STDERR

ERANXO: movi    <$,E$NXOB>,ERRCOD
        jmp     STDERR

ERAUDF: movi    <$,E$UNDF>,ERRCOD
        jmp     STDERR

ERAOPV: movi    <$,E$OBPV>,ERRCOD
        jmp     STDERR

ERAPTT: movi    <$,E$TPRC>,ERRCOD
        popm    r2,r3
        jmp     STDERR

ERAPAC: movi    <$,E$APRC>,ERRCOD
        jmp     STDERR

        endmod  LGOGAR
```

4.2.9 LGOINI.LIL

```
       search  lil

        module  LGOINI — initializes LOGO

        include FLAGS
        include CONSTS
        include TOKENS
; program code generated by this module can be overwritten

;
;       Load tables

;
;format for PRIM generated load table entries
```

```
dname:  ptr     NULL
dflags: int     0
dvalue: ptr     NULL
dsymb:  ptr     NULL
dend:
        Const   NAME,  dname-dname
        Const   FLAGS, dflags-dname
        Const   VALUE, dvalue-dname
        Const   SYMENT, dsymb-dname

        Const   ENTSIZ,dend-dname
;
; externals

        extrn   LEVEL, PARSE, PRVUSR, $PI, RUN, PRVRUN

        vsect

TMPPTR: ptr     null                    ;temporary
TMPINT: int     0

; Primitive Load Table

        extrn   PT              ;Hash table for procedures
                                                ;INSERT$LINE
PRMTAB: prim    INSSTR, 0,      EMPTY,  INSENT  ;not a primitive, but
                                                ;known to PARSER
        prim    MAKSTR, F$PRIM+2,                       MAKE,   MAKENT
        prim    THGSTR, F$PRIM+F$SARG+F$JMP+1,  THING
        prim    DIFSTR, F$PRIM+F$NARG+2,        DIFF,   DIFENT
        prim    PRDSTR, F$PRIM+F$NARG+2,        PROD,   PRDENT
        prim    LNGPRD, F$PRIM+F$NARG+F$GRDY,   PROD
        prim    REMSTR, F$PRIM+F$NARG+2,        REM,    REMENT
        prim    QUOSTR, F$PRIM+F$NARG+2,        DIVIDE, QUOENT
        prim    MINSTR, F$PRIM+F$NARG+1,        MINUS,  MINENT
        prim    LESSTR, F$PRIM+2,               LESSP,  LESENT
        prim    GRTSTR, F$PRIM+2,               GRTRP,  GRTENT
        prim    EQUSTR, F$PRIM+2,               EQUALP, EQUENT
        prim    GRESTR, F$PRIM+2,               GREP
        prim    LSESTR, F$PRIM+2,               LSEP
        prim    IDNSTR, F$PRIM+F$SARG+2,        IDENT
        prim    PCDSTR, F$PRIM+F$SARG+2,        PRCDEP
        prim    FLWSTR, F$PRIM+F$SARG+2,        FOLLWP
        prim    SUMSTR, F$PRIM+F$NARG+2,        SUM,    SUMENT
        prim    LNGSUM, F$PRIM+F$NARG+F$GRDY,   SUM
        prim    PVRSTR, F$PRIM+F$SARG+F$JMP+F$PRIV+1, PRVRUN
        prim    RUNSTR, F$PRIM+F$SARG+F$JMP+1,  RUN
prim    GOSTR,  F$PRIM+F$NARG+F$JMP+1,  GO
prim    DONSTR, F$PRIM+F$JMP+0,         DONE
prim    RETSTR, F$PRIM+F$JMP+1,         RTURN
prim    VALSTR, F$PRIM+F$SARG+F$JMP+1,  VALOF
prim    TRISTR, F$PRIM+F$SARG+F$JMP+2,  TRAPRT
prim    FIRSTR, F$PRIM+F$SARG+1,        FIRST
prim    LASSTR, F$PRIM+F$SARG+1,        LAST
prim    INPSTR, F$PRIM+1,               INTP
prim    ZERSTR, F$PRIM+1,               ZEROP
prim    EPPSTR, F$PRIM+1,               EMPTYP
prim    WRPSTR, F$PRIM+1,               WORDP
prim    LSPSTR, F$PRIM+1,               LISTP
prim    THPSTR, F$PRIM+1,               THINGP
prim    PMPSTR, F$PRIM+1,               PRIMP
prim    PRPSTR, F$PRIM+1,               PROCP
prim    FLPSTR, F$PRIM+1,               FLATP
prim    WRDSTR, F$PRIM+F$SARG+2,        WORD
prim    LWRSTR, F$PRIM+F$SARG+F$GRDY,   WORD
prim    BTFSTR, F$PRIM+F$SARG+1,        BUTFIR
prim    BTLSTR, F$PRIM+F$SARG+1,        BUTLAS
prim    SENSTR, F$PRIM+F$SARG+2,        SENTEN
prim    LSNSTR, F$PRIM+F$SARG+F$GRDY,   SENTEN
prim    LSTSTR, F$PRIM+F$SARG+2,        LIST
prim    LSNLST, F$PRIM+F$SARG+F$GRDY,   LIST
prim    FPTSTR, F$PRIM+F$SARG+2,        FRSPUT
prim    LPTSTR, F$PRIM+F$SARG+2,        LASPUT
prim    PRTSTR, F$PRIM+F$SARG+1,        PRINT
prim    LPRSTR, F$PRIM+F$SARG+F$GRDY,   PRINT
prim    STFSTR, F$PRIM+F$PRIV+F$SARG+F$GRDY+F$QUOT, STUFF
prim    TXFSTR, F$PRIM+F$SARG+1,        TEXFRM
prim    TEXSTR, F$PRIM+F$SARG+1,        TEXTT
prim    NTXSTR, F$PRIM+F$SARG+1,        NOTEXT
prim    TXPSTR, F$PRIM+1,               TEXTP
prim    PRCSTR, F$PRIM+F$PRIV+0,        PROCS
prim    NAMSTR, F$PRIM+F$PRIV+0,        NAMES
prim    IPRSTR, F$PRIM+F$PRIV+0,        IPROCS
prim    RPRSTR, F$PRIM+F$PRIV+0,        PPROCS
prim    INMSTR, F$PRIM+F$PRIV+0,        INAMES
prim    PNMSTR, F$PRIM+F$PRIV+0,        PNAMES
prim    STPSTR, F$PRIM+F$PRIV+F$SARG+2, STTPRC
prim    USPSTR, F$PRIM+F$PRIV+F$SARG+2, USTPRC
prim    STNSTR, F$PRIM+F$PRIV+F$SARG+2, STTNAM
prim    USNSTR, F$PRIM+F$PRIV+F$SARG+2, USTNAM
prim    PPNSTR, F$PRIM+F$PRIV+F$SARG+2, PRPNAM
prim    PPPSTR, F$PRIM+F$PRIV+F$SARG+2, PRPPRC
prim    OPNSTR, F$PRIM+F$SARG+2,        OPEN
prim    CLSSTR, F$PRIM+F$SARG+1,        CLOSE
prim    UOPSTR, F$PRIM+F$SARG+1,        UNOPEN
prim    RENSTR, F$PRIM+F$SARG+2,            RENAME
prim    DELSTR, F$PRIM+F$SARG+2,            DELETE
prim    RDCSTR, F$PRIM+F$SARG+1,            RDCHAR
prim    RDLSTR, F$PRIM+F$SARG+1,            RDLINE
prim    RDSTR,  F$PRIM+F$SARG+1,            READ
prim    TYPSTR, F$PRIM+F$SARG+1,            TYPE
prim    LTYSTR, F$PRIM+F$SARG+F$GRDY,       TYPE
prim    WRTSTR, F$PRIM+F$SARG+F$GRDY,       WRITE
prim    FWTSTR, F$PRIM+F$SARG+F$GRDY,       FWRITE
prim    WTCSTR, F$PRIM+F$SARG+2,            WTCHAR
prim    WTLSTR, F$PRIM+F$SARG+2,            WTLINE
prim    EFLSTR, F$PRIM+F$SARG+1,            ENDFIL
prim    IPCSTR, F$PRIM+F$SARG+1,            INPCHR
prim    FLSSTR, F$PRIM+F$SARG+1,            FILSIZ
prim    ACTSTR, F$PRIM+F$SARG+1,            ACCTYP
prim    CPNSTR, F$PRIM+F$SARG+1,            CHRPTR
prim    SCPSTR, F$PRIM+2,                   STCPTR
prim    COLSTR, F$PRIM+F$SARG+1,            COLUMN
prim    FPRSTR, F$PRIM+F$GRDY+F$SARG,       FPRINT
prim    FTYSTR, F$PRIM+F$GRDY+F$SARG,       FTYPE
prim    PRFSTR, F$PRIM+F$SARG+1,            PRTFRM
prim    PSFSTR, F$PRIM+F$SARG+1,            PARFRM
prim    ASCSTR, F$PRIM+F$SARG+1,            ASCIII
prim    CHRSTR, F$PRIM+F$NARG+1,            CHAR
prim    CNTSTR, F$PRIM+F$SARG+1,            COUNT
prim    SBJSTR, F$PRIM+3,                   SUBOBJ
prim    ELMSTR, F$PRIM+2,                   ELMENT
prim    MEMSTR, F$PRIM+F$SARG+2,            MEMBER
prim    NOTSTR, F$PRIM+F$SARG+1,            NOT
prim    ANYSTR, F$PRIM+F$SARG+F$GRDY,       ANY
prim    ETTSTR, F$PRIM+F$SARG+2,            ANY
prim    ALLSTR, F$PRIM+F$SARG+F$GRDY,       ALL
prim    BOTSTR, F$PRIM+F$SARG+2,            ALL
prim    STMSTR, F$PRIM+F$PRIV+0,            SETMON
prim    UNWSTR, F$PRIM+F$NARG+1,            UNWND
prim    UWWSTR, F$PRIM+2,                   UNWNDW
prim    LEVSTR, F$PRIM+0,                   $LEVEL
prim    DATSTR, F$PRIM+0,                   DATE
prim    TMESTR, F$PRIM+0,                   DAYTIM
prim    VRSSTR, F$PRIM+0,                   VRSNUM
prim    WAISTR, F$PRIM+F$NARG+1,            DELAY
prim    PPTSTR, F$PRIM+F$SARG+1,            POPTRP
prim    TPSSTR, F$PRIM+F$SARG+1,            TRPSET
prim    TPHSTR, F$PRIM+F$SARG+1,            TRPHLR
prim    BKTSTR, F$PRIM+0,                   BAKTRC
prim    SPTSTR, F$PRIM+F$JMP+4,             SPRTRP
prim    STTSTR, F$PRIM+F$SARG+2,            SETTRP
prim    TIMSTR, F$PRIM+0,                   RDTIM
```

```
        prim    STISTR, F$PRIM+F$NARG+1,        SETTIM
        prim    CTISTR, F$PRIM+0,               CLRTIM
        prim    ERPSTR, F$PRIM+F$SARG+1,        ERAPRC
        prim    ERNSTR, F$PRIM+F$SARG+1,        ERANAM
PRMEND:

; System-defined Names Load Table

        extrn   NT                  ;Hash table for names

NAMTAB: prim    EMPSTR, F$NWRT+F$PRIV,  EMPTY
        prim    BOOSTR, F$NWRT+F$PRIV+F$NINV,   BTSSTR
NAMEND:

;
; Reserved word Load table

        extrn   RWT                 ;Hash table for reserved words

RWTTAB: prim    IF,     T$IF
        prim    ELSE,   T$ELSE
        prim    DO,     T$DO
        prim    THEN,   T$DO
        prim    REPT,   T$REPT
        prim    WHI1,   T$WHI1
        prim    UNTL,   T$UNTL
        prim    BCMS,   T$BCMS
        prim    LFSSOP, T$LESS
        prim    GRTROP, T$GRTR
        prim    EQALOP, T$EQAL
        prim    SUMOP,  T$SUM
        prim    DIFFOP, T$DIFF
        prim    PRODOP, T$PROD
        prim    QUOTOP, T$QTNT
        prim    REMOP,  T$REM
RWTEND:

;
; special literal load table

        extrn   SPT

        extrn   APPSTR,CPTSTR,INTSTR,WRTSTR,RDSTR,RNDSTR,ERRSTR
        extrn   NORSTR,GRDSTR,QUTSTR,ENOSTR

SPCTAB: ptr     ERRSTR
        int     F$FILE, IO$ERR, 0
        ptr     CPTSTR
        int     F$FILE, IO$PTR, 0
        ptr     INTSTR
        int     F$OPEN, IO$INT, 0
        ptr     WRTSTR
        int     F$OPEN, IO$WR,  0
        ptr     RDSTR
        int     F$OPEN, IO$RD,  0
        ptr     RNDSTR
        int     F$OPEN, IO$RND, 0
        ptr     APPSTR
        int     F$OPEN, IO$APP, 0
;
;procedure type names

        ptr     NORSTR
        int     F$PTYP, 0,      0
        ptr     QUTSTR
        int     F$PTYP, F$QUOT, 0
        ptr     GRDSTR
        int     F$PTYP, F$GRDY, 0
        ptr     BNOSTR
        int     F$PTYP, F$GRDY+F$QUOT,  0
;
;trap type names

        extrn   PARSTR,EVLSTR,FILSTR,ALCSTR,TRCSTR,CLKSTR,KBDSTR,PANSTR

        ptr     PARSTR
        int     F$TRAP, T$PRSE, 0
        ptr     EVLSTR
        int     F$TRAP, T$EVAL, 0
        ptr     FILSTR
        int     F$TRAP, T$FILE, 0
        ptr     ALCSTR
        int     F$TRAP, T$ALC,  0
        ptr     TRCSTR
        int     F$TRAP, T$TRCE, 0
        ptr     CLKSTR
        int     F$TRAP, T$TIME, 0
        ptr     KBDSTR
        int     F$TRAP, T$KBD,  0
        ptr     PANSTR
        int     F$TRAP, T$PANC, 0
;
; property type names

        extrn   PPRSTR,PINSTR,PTRSTR,PSTSTR,PNRSTR,PNWSTR
        ptr     PPRSTR
        int     F$PPRP+F$NPRP
        ptr     P$PRM
        int     0
        ptr     PINSTR
        int     F$PPRP+F$NPRP
        ptr     P$INV
        int     0
        ptr     PTRSTR
        int     F$PPRP+F$NPRP
        ptr     P$TRC
        int     0
        ptr     PSTSTR
        int     F$PPRP
        ptr     P$STP
        int     0
        ptr     PNRSTR
        int     F$PPRP+F$NPRP
        ptr     P$NRD
        int     0
        ptr     PNWSTR
        int     F$PPRP+F$NPRP
        ptr     P$NWR
        int     0
SPCEND:

;
;  standard file load table

        extrn   FT
        extrn   CSLSTR

FILTAB: ptr     CSLSTR
        int     IO$INT, -1,     0               ;for TOPS-20 ONLY
                                                ;CONSOLE is JFN 0,,-1
        prim    BTSSTR, IO$RD,  ,       BTSENT
FILEND:

;
        pfix    F$STR,16
BOOTLN: text "run read :#boots"

        pfix    F$STR,15
STRTLN: text "startup char 62"

        pfix    F$STR,15
PRIVLN: text "startup char 33"          ;privileged prompt is !
```

```
STLIN:  ptr     STRTLN

PRVLIN: ptr     PRIVLN

INILIN: ptr     BOOTLN

        entry   $INIT,$START,$PRIV

$INIT:  movp    <#,STACK>,r4            ;set up LOGO stack pointer
                                        ; & bottom frame
        movp    r4,<TOP,4>              ;phony TOP entry
        decp    <TOP,4>
        movp    <#,NULL>,<PROC,4>
        movi    <#,-1>,LEVEL
        movi    <#,1>,PRVUSR
        movi    <#,0>,$PI


        movp    <#,PRMTAB>,r1
        movp    <#,PRMEND>,r2
        movp    PT,r3
        call    LOATAB

        movp    <#,NAMTAB>,r1
        movp    <#,NAMEND>,r2
        movp    NT,r3
        call    LOATAB

        movp    <#,RWTTAB>, r1
        movp    <#,RWTEND>, r2
        movp    RWT, r3
        call    LOATAB

        movp    <#,SPCTAB>, r1
        movp    <#,SPCEND>, r2
        movp    SPT, r3
        call    LOATAB

        movp    <#,FILTAB>, r1
        movp    <#,FILEND>, r2
        movp    FT, r3
        call    LOATAB

;
;open bootstrapping file

        movp    <#,BTSSTR>,r1
        movp    <#,IO$RD>,r2
```

421

```
        host    $OPEN
        jmpmi   r0,GIVEUP                ;no open errors permitted
        setval  r0,BTSENT
;
; get initial line & feed it to RUN

        movp    <$,INILIN>,r2
        call    RUN
        return

        extrn   TOPLEV
;
;hot start -- run monitor privileged

$PRIV:  movi    <$,0>,TOPLEV             ;clear TOPLEV from $INIT
        movi    <$,1>,PRVUSR
        movi    <$,0>,$PI
        movp    <$,PRVLIN>,r2
        call    RUN
        return                           ;to host
;
;hot start --run monitor unprivileged (but start up in privileged mode)

$START: movi    <$,0>,TOPLEV             ;clear TOPLEV from $INIT
        movi    <$,0>,PRVUSR
        movi    <$,0>,$PI
        movp    <$,STLIN>,r2
        call    PRVRUN
        return                           ;to host
;
;$BOMB--reenter LOGO after near-fatal storage disaster
;(can't be overwritten)

        extrn   UNWIND, GETRST

        psect

        entry   $BOMB

$BOMB:  movi    TOPLEV,r1                ;get top user frame of LOGO
        call    UNWIND
        call    GETRST                   ;in EVAL, restores context
        return                           ; & continues

        vsect
;
;[
;routine: LOATAB --loads a symbol table from a PRIM table
;input regs: r1 —start of load table
;            r2 —end of load table
;            r3 —ptr to symbol table
;CALLS:
        extrn   GIVEUP
;]

ADRENT: ptr     NULL                    ;temp. loc. for addr. of entry

LOATAB: subvec  r3,<#,TMPINT>,<#,1>     ;get length of hash table
        movi    TMPINT,r0
        movp    <#,NULL>,TMPPTR
;
;fill hash table with NULLs

LOATA2: jmpli   r0,<#,2>,LOATA4
        rplvec  <#,TMPPTR>,r3,r0
        deci    r0
        jmp     LOATA2
;
;make entries in symbol table

LOATA4: movp    <SYMENT,1>,r0           ;get location to put addr. of entry
        jmpnep  r0,<#,0>,LOATA5
        movp    <#,ADRENT>,r0           ;default is to put addr. of entry
                                        ;in temp. location
LOATA5: enter   r3,<NAME,1>,<@,0>,GIVEUP
        setflg  <FLAGS,1>,<@,0>
        setval  <VALUE,1>,<@,0>
        addp    <#,ENTSIZ>,r1
        jmplp   r1,r2,LOATA4

        return

        endmod  LGOINI
```

4.2.10 LGOIO.LIL

```
       search LIL

        module LGOIO    ; LOGO I/O primitives & support

        include FLAGS
        include CONSTS
        include ERRORS
        include TOKENS

; use string buffers defined in LGOSTR

        extrn SBISIZ,SBICNT,SBIPTR,SBIOBP,SBIOBC,SBIOBL,SBIFLG,SBIBUF
        extrn SBOOBP,SBOOBC,SBOFLG,SBOBUF
;
;external routines

        extrn BININI,BINP,SETINP,CMDDON,STDERR,RDOBJ,CHTYPE
        extrn BOUINI,BOUP,BOUEND,CRNUM,CVTNU0,CVTSTR,GIVEUP
;
; external symbol tables

        extrn   SPT, FT
;
; external variables

        extrn   EMPTY,TRUEP,FALSP,$PI

;
; external strings (for use in ACCTYP)

        extrn   INTSTR,RDSTR,WRTSTR,APPSTR,RNDSTR,UPNSTR
;
; error variables

        extrn   ERRCOD,ERROBJ,ERRINF

;temporaries

        vsect

NOEOL:  int     0                       ; flag for no EOL between output
                                        ; items in WTARGS, WTCARG code
                                        ; (or no EOL trans. in GIVCHR)
FULTRN: int     0                       ; flag for object translation forms
MODE:   int     0                       ; access mode of current file
TMPCHR: int     0
FD:     int     0                       ; file descriptor of current file
TMPOBJ: ptr     null                    ; (possibly indirect) ptr. to
                                        ; current output or input object
WSTART: int     0                       ; offsets in output object to start
WSTOP:  int     0                       ; and stop writing
LINSIZ: int     0                       ;width of line of file (0 is
                                        ;infinite width)
```

```
BLKFLG: int     0                       ; flag for non-blank char
                                        ; found in a word-chunk.
OBJLEN: int     0                       ; length of wd. passed to OUTOBJ
        psect
;
;       I/O support routines

;[
;routine: GETOBJ — BCB routine to give object to buffer management
;output reg: r0
;attr: SAFE
;]

GETOBJ: movp    TMPOBJ,r0
        return

;[
;routine: GETIND — BCB routine to give object to buffer management
;output reg: r0
;attr: SAFE
;]

GETIND: movp    TMPOBJ,r0
        movp    <@,0>,r0
        return

;[
;routine: TRNOBJ — translates a LOGO object into a string corresponding
;                 to the form in which it would be printed.
;input regs:      FULTRN gives the type of print form to be used; if zero,
;                 it means translate as per PRINT; if positive, it means
;                 create a FULLPRINT-style string, which looks like the
;                 form in which the object would be read in via RDOBJ.
;                 if negative, translate into TEXTFORM string,(the form in which
;                 procedure text is SHOWn)
;                 r1 is the pointer to the object to be translated.
;output regs: r0 contains a pointer to the resulting string.
;attr: UNSAFE
;called by: PRTFRM, TYPOBJ, WRITE
;globals: FULTRN
;calls: buffer management, CHTYPE
;]
        entry   TRNOBJ

TRNOBJ: pushm   r2,r3
        movp    r1,TMPOBJ               ; save (indirect)address of object
                                        ;EDITED by RJS
;  initialize buffers
```

```
        movp    <#,GETIND>,SBIOBP
        movi    <#,1>,SBIOBC
        movi    <#,B$STR>,SBIFLG
        movp    <#,SBIBUF>,r1
        call    BININI
        swapp   <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI
;
; insert opening bracket if obj. is a list and we are not in PRINT mode.

        movp    TMPOBJ,r0
        gettyp  <@,0>,r3
        jmpzi   FULTRN,TRNLP            ; don't check type for PRINT mode
        tstb    <#,F$LIST>,r3,TRNLLB
        jmp     TRNLP
TRNLLB: movp    <#,SBOBUF>,r1
        movi    <#,"[">,r2
        call    BOUP
;
; main translation loop

TRNLP:  movp    <#,SBIBUF>,r1
        call    BINP                    ; get a character from object
        tstb    <#,B$DONE>,SBIFLG,TRNDON ; quit loop if no more chars
        jmpei   <#,U$LBRK>,r0,TRNULB
        jmpei   <#,U$RBRK>,r0,TRNURB    ; translate untouchables
        jmpei   <#,U$SPCS>,r0,TRNUSP

        jmpmi   FULTRN,TRNBOU           ;no other translation for TEXTFORM

        jmpei   <#,"#">,r0,TRNHSH       ; maybe translate a poundsign
        jmpzi   FULTRN,TRNBOU   ; no @'s printed unless FULL mode
        jmpei   <#,SPACE>,r0,TRNQUO     ; a space always needs an @
        jmpei   <#,"[">,r0,TRNQUO       ; as do brackets
        jmpei   <#,"]">,r0,TRNQUO
        tstb    <#,F$LIST>,r3,TRNBOU    ; and, if object isn't a list,
        call    CHTYPE                  ;  so do all punctuation chars
        tstb    <#,T$PUNC>,r1,TRNQUO
;
;  output a character of the translated object

TRNBOU: movi    r0,r2
TRNBO1: movp    <#,SBOBUF>,r1
        call    BOUP
        jmp     TRNLP                   ; loop for next char
```

```
;
; code to handle untouchables

TRNULB: movi    <#,"[">,r2
        jmp     TRNB01
TRNURB: movi    <#,"]">,r2
        jmp     TRNB01
TRNUSP: movi    <#,SPACE>,r2
        jmp     TRNB01

;
;  output a hash mark only in FULLPRINT mode, otherwise a space.

TRNHSH: jmpnzi  FULTRN,TRNBOU
        jmp     TRNUSP
;
;  output the character in r0 preceded by an "@".

TRNQUO: pushi   r0
        movi    <#,"@">,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        popi    r2
        jmp     TRNB01

TRNDON: jmpzi   FULTRN,TRNRET           ; add trailing bracket if list
        tstb    <#,F$LIST>,r3,TRNLRB    ; and not in PRINT mode
        jmp     TRNRET
TRNLRB: movp    <#,SBOBUF>,r1
        movi    <#,"]">,r2
        call    BOUP

TRNRET: movp    <#,SBOBUF>,r1
        call    BOUEND
        cmovp   SBOOBP,r0
        popm    r2,r3
        return
;
;[
;routine: OUTOBJ -- writes a LOGO word to an output file. If the file's
;               line-width is zero, the word is written out unchanged,
;               byte-by-byte. If the line-width is non-zero, EOL's may
;               be inserted between contiguous chunks of printing chars
;               in order to chop up the word into line-sized pieces.
;input regs: r1 -- File Descriptor. r2 -- string ptr.
;output regs: r0 destroyed
;attr: SAFE
;type :utility
```

```
;calls: BUFFER MANGEMENT
;globals: String BCB's, FD, TMPOBJ, WSTART, WSTOP
;]
        entry   OUTWRD

OUTWRD: push    r3
        jmpep   r2,<#,EMPTY>,OUTDON     ; don't bother outputting EMPTY
        movi    r1,FD
        movp    r2,TMPOBJ               ; store obj. ptr. (might be transient)
        getlen  r2,OBJLEN               ; store its length as well

        movi    <#,IO$WID>,r2
        host    $SENSE                  ; get width of line
        movi    r0,LINSIZ

        movp    <#,GETOBJ>,SBIOBP       ; give routine addr. to buffer mgr.
        movi    <#,1>,SBIOBC            ; start at begining of object
        movi    <#,B$STR>,SBIFLG
        movp    <#,SBIBUF>,r1
        call    BININI                  ; init. BCB
        movi    <#,1>,WSTART
        jmpzi   LINSIZ,OUTWDZ           ; output whole word if no width
        movi    <#,0>,WSTOP
;
;  look for the next chunk of the word. A chunk is terminated by a
;  "white space"..i.e. a tab, space, or EOL. Chunks may, however, begin
;  with any number of consecutive "white spaces".

OUTWDL: movp    <#,SBIBUF>,r1
        movi    <#,0>,BLKFLG            ; no non-blanks encountered yet

OUTWD1: call    BINP
        tstb    <#,B$DONE>,SBIFLG,OUTCHK
        jmpei   r0,<#,SPACE>,OUTWDU     ; check for white space
        jmpei   r0,<#,TAB>,OUTWDU
        jmpei   r0,<#,EOL>,OUTWDU
        movi    <#,1>,BLKFLG            ; signal non-blank found
        jmpnei  r0,<#,"@">,OUTWD3      ; if character quote, skip
        inci    WSTOP
        call    BINP                    ;...over quoted character
OUTWD3: inci    WSTOP
        jmp     OUTWD1
OUTWDU: jmpzi   BLKFLG,OUTWD3           ; don't stop on leading blank.
        jmp     OUTCHK          ; if non-leading, write a chunk!
;
;  come here if writing from WSTART to end of word.

OUTWDZ: movi    OBJLEN,WSTOP
```

```
;
; write out a chunk if non-leading white space (or end of word) found.

OUTCHK: movi    WSTART,r2       ; set input buffer to start of unit
        call    SETINP
        jmpzi   LINSIZ,OUTLP1   ; don't check col. if zero linewidth

        movi    LINSIZ,r3

        movi    FD,r1
        movi    <#,IO$COL>,r2           ; get current column position
        host    $SENSE
        subi    r0,r3                   ;   in r0
                                ; (r3 has # of columns left on line)

        movi    WSTOP,r0
        subi    WSTART,r0               ; r0 has length of unit
        inci    r0
        jmpgi   r3,r0,OUTLP             ; jump if unit fits on line
        jmpei   r3,LINSIZ,OUTLP         ; and we aren't at right margin,
        call    OUTLIN                  ; else output EOL
        movi    LINSIZ,r3

OUTLP:  jmpzi   LINSIZ,OUTLP1           ; write right away if zero width
        jmppi   r3,OUTLP1               ; if no room left on line
        call    OUTLIN                  ; output EOL
        movi    LINSIZ,r3

OUTLP1: movp    <#,SBIBUF>,r1
        call    BINP                    ; get char. from string
        movi    FD,r1
        movi    r0,r2
        host    $WRCH
        inci    WSTART
        deci    r3
        jmpngi  WSTART,WSTOP,OUTLP
        jmpli   WSTOP,OBJLEN,OUTWDL     ; quit if object finished

;
;return from OUTOBJ

OUTDON: pop     r3
        return

;[
;routine: OUTLIN -- outputs EOL sequence to file
;]
```

```
OUTLIN: movi    FD,r1
        movi    <#,EOL>,r2
        host    $WRCH
        return
;
;[
;routine: WRITE -- primitive for writing an unspecified # of LOGO
;               objects to an output file.
;input regs: std. prim.
;output regs: r0 UNDEF always
;attr: UNSAFE
;]

        entry   WRITE

WRITE:  movi    <#,0>,FULTRN            ; non-full translation
        movi    <#,0>,NOEOL             ; EOL between arguments
        jmp     WTFILE          ; go to WRITE/FWRITE common code

;[
;routine: FWRITE -- FULLWRITE primitive
;                       same as WRITE, but sets FULTRN to 1.
;]
        entry   FWRITE

FWRITE: movi    <#,1>,FULTRN
        movi    <#,0>,NOEOL
;
; entry point for WRITE

WTFILE: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3            ; look up file name
        jmpep   <#,NULL>,r3,IOERR1      ; error if not open
        getflg  r3,r0                   ; get access mode
        jmpei   r0,<#,IO$RD>,IOERR2     ; error if read mode
        deci    r1                      ; point to first data arg.
        decp    r2
;
; write the remainder of the argument list
; (entry for PRINT, FULLPRINT)

WTARGS: getval  r3,FD                   ; r3 has index into filename table
        pop     r3
WTLOOP: jmpzi   r1,CMDDON
        pushm   r1,r2
        movp    r2,r1
        call    TRNOBJ
        cmovp   r0,r2
        pushp   r2
        movi    FD,r1
        call    OUTWRD
        popp    r2
        swapp   <#,NULL>,r2       ; deallocate translated argument string
        jmpnzi  NOEOL,WTNEOL              ; check line-skip flag
        movi    FD,r1                     ; if off, insert EOL between args.
        call    OUTLIN
WTNEOL: popm    r1,r2
        deci    r1
        decp    r2
        jmp     WTLOOP
;
        extrn   CSLSTR                    ; address of console keyword

;
;  come here after setting appropriate mode words for terminal
;  output primitives.

WRTCON: push    r3
        find    FT,<#,CSLSTR>,r3
        jmp     WTARGS

;[
;routine: PRINT
;input regs: std. prim.
;output regs: r0 —UNDEF always
;attr: SAFE
;]
        entry   PRINT

PRINT:  movi    <#,0>,FULTRN              ; non-full translation
        movi    <#,0>,NOEOL               ; skip between arguments
        jmp     WRTCON

;[
;routine: TYPE (temporary version)
;input regs: std. prim.
;output regs: r0 —UNDEF always
;attr: SAFE
;]
        entry   TYPE

TYPE:   movi    <#,0>,FULTRN
        movi    <#,1>,NOEOL               ; as PRINT, but no skip
        jmp     WRTCON
```

```
;[
;routine: FPRINT — FULLPRINT
;input regs: std. prim.
;output regs: r0 —UNDEF always
;attr: SAFE
;]
        entry   FPRINT

FPRINT: movi    <#,1>,FULTRN            ; Full translation
        movi    <#,0>,NOEOL             ; skip between arguments
        jmp     WRTCON

;[
;routine: FTYPE — FULLTYPE primitive
;input regs: std. prim.
;output regs: r0 —UNDEF always
;attr: SAFE
;]
        entry   FTYPE

FTYPE:  movi    <#,1>,FULTRN
        movi    <#,1>,NOEOL             ; as PRINT, but no skip
        jmp     WRTCON
;
;[
;routine: PRTFRM — #PRINTFORM primitive
;input regs: std. primitive
;output regs: r0 points to FULLPRINT print form of object given as arg.
;attr: UNSAFE
;]
        entry   PRTFRM

PRTFRM: movi    <#,1>,FULTRN
        movp    r2,r1
        call    TRNOBJ
        succeed
;[
;routine: TEXFRM — #TEXTFORM primitive
;input regs: std. primitive
;output regs: r0 points to textual form of object given as arg.
;attr: UNSAFE
;]
        entry   TEXFRM

TEXFRM: movi    <#,-1>,FULTRN
        movp    r2,r1
        call    TRNOBJ
        succeed
```

```
;
;[
;routine: WTCHAR (WRITECHAR operation)
;input regs: std. normal prim.
;               1st arg. is file name
;               2nd arg is a word.
;output reg: r0 --UNDEF
;calls: $RDCH
;]
        entry   WTCHAR

WTCHAR: movi    <#,1>,NOEOL
        jmp     WTCARG

;[
;routine: WTLINE (WRITELINE operation)
;input regs: std. normal prim.
;               1st arg. is file name
;               2nd arg is a word.
;output reg: r0 --UNDEF
;calls: $RDCH
;]
        entry   WTLINE

WTLINE: movi    <#,0>,NOEOL


; WTCARG -- does the work for WTLINE/WTCHAR primitives

WTCARG: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3             ;look up file name
        jmpep   <#,NULL>,r3,IOERR1       ;jump if file not open
        getflg  r3,r0                   ;get access mode
        jmpei   r0,<#,IO$RD>,IOERR2     ; error unless write access
;
; write char.

        getval  r3,r1                   ;get FD
        decp    r2                      ; point to character arg
        jmpep   <@,2>,<#,EMPTY>,WTCHA2
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3    ; error if it's a list
        getlen  <@,2>,WSTOP
        movp    r2,r3
        movi    <#,1>,WSTART
WTCHA1: substr  <@,3>,<#,TMPCHR>,WSTART,<#,1>
```

```
        movi    TMPCHR,r2
        host    $WRCH                   ; write that byte
        inci    WSTART
        jmpngi  WSTART,WSTOP,WTCHA1     ; return when no more chars

; write extra EOL if nexessary

WTCHA2: jmpnzi  NOEOL,WTCDON
        movi    r1,FD
        call    OUTLIN

WTCDON: pop     r3
        jmp     CMDDON
;

;
;[
;routine :OPEN --primitive to open file
;input regs: std. normal prim.
;               1st is implementation-dependent file specification
;               2nd arg. is file access type literal
;output reg: std. command
;]
        entry   OPEN

OPEN:   push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        decp    r2
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    SPT,<@,2>,r0           ;look up literal
        jmpep   r0,<#,NULL>,OPERR1      ;jump if literal not found
        getflg  r0,r3
        tstb    <#,F$OPEN>,r3,OPEN1     ;check if this literal belongs
        jmp     OPERR1                  ; to OPEN

OPEN1:  incp    r2                      ;get 1st arg.
        movp    <#,NULL>,r1
        movs    <@,2>,r1               ;get copy of file name
        enter   FT,r1,r3,OPERR2         ;create entry in file table
                                        ;error if entry already exists
        getval  r0,r0                   ;get access type code
        setflg  r0,r3                   ;store in table
        pushp   r2
        movp    <@,2>,r1
        movi    r0,r2
        host    $OPEN
        popp    r2
        jmpmi   r0,OPERR3               ;negative FD returned signals error
        setval  r0,r3                   ;value of FT entry is FD
        pop     r3
        jmp     CMDDON                  ;std. command return
;
; unknown access type error

OPERR1: movi    <#,E$UNAT>,ERRCOD
        oop     r3
        jmp     STDERR                  ;std. failure
;
; file already opened error

OPERR2: movi    <#,E$FAOP>,ERRCOD
        swapp   <#,NULL>,r1             ;dealc. copy of file name
        pop     r3
        jmp     STDERR
;
; system open error

OPERR3: movi    <#,E$OFAI>,ERRCOD       ;open failure
        remove  FT,<@,2>               ;deallocate file table entry
                                        ;  for erroneous file
        pop     r3
        jmp     STDERR
;
;[
;routine: CLOSE -- primitive to close file
;input regs: std. primitive
;               1st arg. is file name
;output reg: r0--UNDEF
;]

        entry   CLOSE

CLOSE:  push    r3
        movi    <#,IO$CLS>,r3           ; pass close disp. to common code
        jmp     CLOSFD

;[
;routine: DELETE
;       primitive to clse and delete a file from permanent storage.
;       requires that the file be open.
;]

        entry   DELETE
```

```
DELETE: push    r3
        movi    <$,IO$DEL>,r3
        jmp     CLOSFD

;[
;routine: UNOPEN
;         primitive to close and possibly abort a file; otherwise
;         the same as CLOSE.
;]

        entry   UNOPEN

UNOPEN: push    r3
        movi    <$,IO$ABT>,r3


; CLOSFD — common code for CLOSE and UNOPEN to close an FD

CLOSFD: gettyp  <@,2>,r0
        tstb    <$,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r1             ; lookup filename
        jmpep   <$,NULL>,r1,IOERR1       ; error if not open
        remove  FT,<@,2>                ; deallocate file table entry
        getval  r1,r1                    ; get FD in r1
        movi    r3,r2
        host    $CLOSE                   ; and do a normal close
        pop     r3
        jmp     CMDDON

;[
;routine: RENAME
;       —primitive to rename a file.  accepts two arguments; the first
;       must be the name of a currently open file; the second a filename
;       to which the open file will be renamed. The file will be closed
;       in the process of renaming.
;input regs: std. primitive
;attr: SAFE
;]

        entry   RENAME

RENAME: push    r3
        decp    r2
        gettyp  <@,2>,r0                ; error if dest. filename is list
        tstb    <$,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3             ; look up dest. filename
        jmpnep  <$,NULL>,r3,RNERR1       ; error if currently open
        incp    r2
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3     ; error if filename is a list
        find    FT,<@,2>,r1             ; look up filename
        jmpep   <#,NULL>,r1,IOERR1       ; error if source file not open
        remove  FT,<@,2>
        getval  r1,r1
        pushp   r2
        movi    <#,IO$CLS>,r2
        host    $CLOSE                   ; close source file f.d.
        popp    r2
        movp    <@,2>,r1                ; move string ptr. to r1
        decp    r2
        movp    <@,2>,r2                ; dest filename ptr. to r2
        host    $RENAME                  ; and do it
        pop     r3
        jmp     CMDDON

RNERR1: movi    <#,E$FAOP>,ERRCOD
        pop     r3
        jmp     STDERR
;
;[
;routine: RDCHAR (READCHAR operation)
;input regs: std. normal prim.
;               1st arg. is file name
;output reg: r0 -- returns 1 char. word. read from file
;calls: $RDCH
;]
        entry   RDCHAR

RDCHAR: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r1             ;look up file name
        jmpep   <#,NULL>,r1,IOERR1       ;jump if file not open
        getflg  r1,r0                    ;get access mode
        jmpei   r0,<#,IO$INT>,RDCHA1     ;read if: interactive
        jmpei   r0,<#,IO$RND>,RDCHA1     ;         random access
        jmpei   r0,<#,IO$RD>,RDCHA1      ;         read access
        jmp     IOERR2                   ;otherwise error
;
; get char.

RDCHA1: getval  r1,r1                    ;get FD
        host    $RDCH
        jmpnzi  $PI,RDINT                ;check for interrupt while in
                                         ; IO wait
        movi    r0,TMPCHR                ;save char.
```

```
;
; allocate 1 char. string & store in word

        movi    <#,1>,r1
        movi    <#,F$STR>,r2
        movp    <#,TMPOBJ>,r3
        movp    <#,NULL>,TMPOBJ          ;clear junk in TMPOBJ
        host    $ALLOC
        rplstr  <#,TMPCHR>,TMPOBJ

        omovp   TMPOBJ,r0
        pop     r3
        succeed
;
; interrupted

RDINT:  pop     r3
        fail
;
;[
;routine: RDLINE (READLINE operation)
;input regs: std. normal prim.
;               1st arg. is file name
;output reg: r0 — returns 1 char. word. read from file
;calls: $RDCH
;]
        entry   RDLINE

RDLINE: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r1             ;look up file name
        jmpep   <#,NULL>,r1,IOERR1       ;jump if file not open
        getflg  r1,r0                    ;get access mode
        jmpei   r0,<#,IO$INT>,RDLIN1     ;read if: interactive
        jmpei   r0,<#,IO$RND>,RDLIN1     ;         random access
        jmpei   r0,<#,IO$RD>,RDLIN1      ;         read access
        jmp     IOERR2                   ;otherwise error
;
; init. output buffer

RDLIN1: getval  r1,FD
        swapp   <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI
        movi    <#,1>,NOEOL              ; no EOL trans. by GIVCHR
;
```

```
; read up to EOF or EOL

RDLIN2: call    GIVCHR,RDLIN3
        jmpei   <#,EOL>,r0,RDLIN3
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        jmp     RDLIN2

RDLIN3: movp    <#,SBOBUF>,r1
        call    BOUEND                  ; return line object
        omovp   SBOOBP,r0
        pop     r3
        succeed

;
;[
;routine: READ
;input regs: std. normal prim.
;               1st arg. is file name
;output reg: r0 — returns next object (word or list) from file
;                  EOL's are considered spaces
;calls: $RDCH, $SENSE, RDOBJ
;]
;
;GIVCHR --feeds characters to reader, fails on EOF

GIVCHR: pushm   r1,r2
        movi    FD,r1
        host    $RDCH
        jmpei   MODE,<#,IO$INT>,GIVCH1  ; skip EOF check for interactive files

        movi    r0,TMPCHR
        movi    <#,IO$EOF>,r2
        host    $SENSE
        jmpnzi  r0,GIVEOF               ; $SENSE returns 1 if EOF
        movi    TMPCHR,r0

GIVCH1: popm    r1,r2
        jmpnzi  NOEOL,GIVEND            ; if NoEoL is on,
        jmpnei  r0,<#,EOL>,GIVEND       ; translate EOL to space
        movi    <#,SPACE>,r0
GIVEND: succeed

GIVEOF: popm    r1,r2
        fail

        entry   READ
```

```
READ:   push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3               ;look up file name
        jmpep   <#,NULL>,r3,IOERR1         ;jump if file not open
        getflg  r3,r0                      ;get access mode
        jmpei   r0,<#,IO$INT>,READ1        ;read if: interactive
        jmpei   r0,<#,IO$RND>,READ1        ;          random access
        jmpei   r0,<#,IO$RD>,READ1         ;          read access
        jmp     IOERR2                     ;otherwise error
;
; RDOBJ does reading

READ1:  getval  r3,FD
        movi    r0,MODE                    ; save access mode
        movi    <#,0>,NOEOL                ; GIVCHR to translate EOL's
        movp    <#,GIVCHR>,r1              ; routine to give characters to reader
        call    RDOBJ
        jmpep   r0,<#,UNDEF>,RDERR1        ; return UNDEF signals ill-formed list
        pop     r3
        succeed
;
; ill-formed list read

RDERR1: movi    <#,E$IOBL>,ERRCOD
        pop     r3
        jmp     STDERR

;
;[
;routine: ENDFIL -- EndFile? primitive, returns boolean TRUE if file
;                 pointer is past EOF.
;input regs: std. primitive
;       1st arg -- file name
;output regs: r0
;]

        entry   ENDFIL

ENDFIL: push    r3
        gettyp  <@,2>,r0                   ; error if filename is list
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3
        jmpep   <#,NULL>,r3,IOERR1
        movi    <#,IO$EOF>,r2
SENSTF: getval  r3,r1
        pop     r3
        host    $SENSE
        jmpzi   r0,SENSF                ; if SENSE returns nonzero,
        movp    <#,TRUEP>,r0            ; then return TRUE
        succeed
SENSF:  movp    <#,FALSP>,r0            ; otherwise we're not.
        succeed

;[
;routine: INPCHR — Input? primitive, returns TRUE if character is
;                waiting on specified interactive file
;input regs: std. prim.
;output regs: r0
;]

        entry   INPCHR

INPCHR: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3    ; error if filename is list
        find    FT,<@,2>,r3
        jmpep   <#,NULL>,r3,IOERR1
        getflg  r3,r0                   ; error if non-interactive file
        jmpnei  r0,<#,IO$INT>,IOERR2
        movi    <#,IO$CHR>,r2
        jmp     SENSTF
;[
; routines: FILSIZ, CHRPTR, COLUMN
;       code for FileSIze, CharPointer, and Column primitives
; input regs: standard primitive
;       1st arg—filename
; output regs: r0— points to a number whose value is the file
;                       attribute asked for.
;]

        entry   FILSIZ,CHRPTR,COLUMN

FILSIZ: movi    <#,IO$SIZ>,r0
        jmp     SENINT
CHRPTR: movi    <#,IO$PTR>,r0
        jmp     SENINT
COLUMN: movi    <#,IO$COL>,r0

; SENINT: common code for returning numbers given by $SENSE

SENINT: push    r3
        gettyp  <@,2>,r1
        tstb    <#,F$LIST>,r1,IOERR3
        find    FT,<@,2>,r3
        jmpep   <#,NULL>,r3,IOERR1
        getval  r3,r1
        pop     r3
        movi    r0,r2                   ; set up appropriate item
        host    $SENSE
        jmpzi   r0,SENIN0               ; if result is nonzero,
        movi    r0,TMPCHR               ; allocate a 2-word number
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,TMPCHR>,r0,<#,2>
        succeed                         ; and put it there.
SENIN0: movi    <#,1>,r1                ; otherwise, allocate a zero
        call    CRNUM                   ; number-vector.
        succeed

;
;[
;routine: ACCTYP -- AccessType primitive
;input regs: std. prim.
;               1st arg -- filename
;output regs: r0 points to the Access Type-code for the file
;]

        entry   ACCTYP

ACCTYP: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        movp    <#,NULL>,r0
        find    FT,<@,2>,r3
        jmpep   <#,NULL>,r3,ACCTUO
        getflg  r3,r1                   ; get access code in r1
        jmpnei  <#,IO$RD>,r1,ACCT01     ; and get the corresponding str.
        movs    <#,RDSTR>,r0            ; in r0
        jmp     ACCDON
ACCT01: jmpnei  <#,IO$WR>,r1,ACCT02
        movs    <#,WRTSTR>,r0
        jmp     ACCDON
ACCT02: jmpnei  <#,IO$INT>,r1,ACCT03
        movs    <#,INTSTR>,r0
        jmp     ACCDON
ACCT03: jmpnei  <#,IO$APP>,r1,ACCT04
        movs    <#,APPSTR>,r0
        jmp     ACCDON
ACCT04: movs    <#,RNDSTR>,r0
        jmp     ACCDON
ACCTUO: movs    <#,UPNSTR>,r0
ACCDON: pop     r3
```

```
        succeed

;
;[
;routine: STCPTR -- SetCharPointer primitive
;input regs: std. prim.
;               1st arg-- file name
;               2nd arg-- integer character pointer
;output regs: r0, UNDEF
;]

        entry   STCPTR

STCPTR: push    r3
        movi    <#,1>,r1
        call    CVTSTR,GIVEUP
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,IOERR3
        find    FT,<@,2>,r3
        jmpep   <#,NULL>,r3,IOERR1
        getval  r3,r3
        decp    r2                      ; point to pointer arg.
        call    CVINU0,STCERR           ; make sure it's numeric
        movi    r3,r1
        subvec  <@,2>,<#,TMPCHR>,<#,1>
        jmpnei  TMPCHR,<#,1>,STCPT1     ; jump if it's nonzero
        movi    <#,0>,r3        ; (if it's 1-word vec., must be zero)
        jmp     STCPT2
STCPT1: subvec  <@,2>,<#,TMPCHR>,<#,2>
        movi    TMPCHR,r3       ; use 2nd word if it's 2-word vec
STCPT2: movi    <#,IO$PTR>,r2
        host    $CONTROL
        pop     r3
        jmp     CMDDON

STCERR: pop     r3              ; come here if CVINU0 fails
        fail


;
;       General IO error returns
;
; attempt to access unopened file

IOERR1: movi    <#,E$FNOP>,ERRCOD
        pop     r3
        jmp     STDERR
;
```

```
; attempt to access file in incorrect mode

IOERR2: movi    <$,E$FINC>,ERRCOD
        pop     r3
        jmp     STDERR
;
; word found instead of list (usually the result of a bad filename)

IOERR3: pop     r3
        movi    <$,E$WELF>,ERRCOD
        jmp     STDERR

        endmod  LGOIO
```

4.2.11 LGONAM.LIL

```
       search LIL

        module LGONAM -- LOGO symbol name table primitives

        include FLAGS
        include CONSTS
        include ERRORS

        extrn   ERRCOD                  ; Error code (integer)
        extrn   ERRINF                  ; error contxt
        extrn   ERROBJ                  ; copy of offending Argument

        extrn   NT,PT,SPT               ;name,procedure, special literal tables
;
; external utility routines & variables

        extrn   STDERR                  ;std. error return
        extrn   CMDDON                  ;std. command return
        extrn   EMPTY, REMPTY, PRVUSR
;
;[
;routine: MAKE -- makes 2nd arg. the "thing" (value) of the 1st
;input regs: std. normal primitive (r1 ignored)
;output reg: r0
;attr: UNSAFE
;]
        entry MAKE
```

```
MAKE:   gettyp  <@,2>,r0                ;get flags of 1st arg.
        tstb    <#,F$LIST>,r0,MAKERR     ;must be a word
        tstb    <#,F$NUM>,r0,MAKER3      ;& not a number
        enter   NT,<@,2>,r0             ;get or create name table
                                         ;entry (no error jump)
;
;test for protected names

        getflg  r0,r1
        tstb    <#,F$NWRT>,r1,MAKER2     ;write-protected

        decp    r2                       ;get 2nd arg.
        setval  <@,2>,r0                ;store in name table
        jmp     CMDDON                   ;std. command return

MAKERR: movi    <#,E$WELF>,ERRCOD        ;word expected,list found
        jmp     STDERR                   ;standard failure

MAKER2: movi    <#,E$OBPV>,ERRCOD
        jmp     STDERR

MAKER3: movi    <#,E$WENF>,ERRCOD
        jmp     STDERR

;
;[
;routine: NEXT -- gets nexts arg. to STUFF, fails if no more args.
;output: ARGCNT,r2 decremented
;attr: SAFE
;globals: ARGCNT
;]

NEXT:   jmpnpi  ARGCNT,NXTERR
        deci    ARGCNT
        decp    r2
        succeed

NXTERR: fail

;[
;routine: STUFF -- (privileged LOGO command #STUFF)
;               creates procedure directory & table entry from
;               LOGO objects
; format of STUFF args:
; PROCEDURENAME PROCEDURETYPE NUMBEROFARGS NUMBEROFLOCS NUMBEROFLINES
;       ARG1 ... ARGN  LOC1 ... LOCN
;       FIRSTLINENO [ source code for line ]
;               .
;               .
;               .
;       LASTLINENO [ source code for last line ]
;
;input reg: std. primitive
;output reg: std. command
;attr: UNSAFE
;calls: NEXT,CMDDON
;]
;
; temp. vars. for both STUFF & TEXTT
        vsect

PRCNAM: ptr     NULL                    ;procedure name
PRCDIR: ptr     NULL                    ;procedure directory
PRCENT: ptr     NULL                    ;procedure entry
PRCFLG: int     0                       ;proc. entry flags
PDFLAG: int     0                       ;proc. dir. flags
NINTS:  int     0                       ;1st 4 words of proc. dir.
NLOCS:  int     0
NARGS:  int     0
NLINES: int     0

LINCNT: int     0                       ;temp. to count lines in proc.
LINOFF: int     0                       ;offset to line numbers
SRCOFF: int     0                       ;offset to source code
ARGCNT: int     0
TMPINT: int     0
TMPPTR: ptr     NULL

        psect

        const   F$PROC,F$PRIM+F$PRIV    ;proc. is LOGO or LIL primitive

        entry STUFF

STUFF:  push    r3
;
; name of procedure is first arg.

        jmpzi   r1,STFERR
        deci    r1
        movi    r1,ARGCNT               ;ARGCNT will have # of args. remaining
        movi    <#,0>,PRCFLG            ;these will be default flags for the
        movi    <#,0>,PDFLAG            ; new proc. dir. & entry
        enter   PT,<@,2>,PRCENT,STUFFA ;look up procedure name
        jmp     STUFFC                  ; table entry created
```

```
;
;procedure already exists

STUFFA: getflg  PRCENT,PRCFLG
        tstb    <#,F$PRIM>,PRCFLG,STFER2 ;proc. privilege violation
        jmpnzi  PRVUSR,STUFFB
        tstb    <#,F$PRIV>,PRCFLG,STFER2 ;proc. privilege violation
STUFFB: getval  PRCENT,r0                ;get old proc. dir.
        jmpep   r0,<#,UNDEF>,STUFFC      ;proc. is erased
        gettyp  r0,PDFLAG
        tstb    <#,F$PWRT>,PDFLAG,STFER2 ;proc. is no change
;
;set flags for new procedure

STUFFC: clrb    <#,F$CHGT>,PRCFLG        ;clear flags which don't survive
        clrb    <#,F$CHGD>,PDFLAG        ;after redefinition
;
; get type of procedure

        call    NEXT,STFERR
        find    SPT,<@,2>,r1            ;look up procedure type literal
        jmpep   <#,NULL>,r1,STFERR       ;not found
        getflg  r1,r3
        tstb    <#,F$PTYP>,r3,STUFF0     ;check if proper literal
        jmp     STFERR
STUFF0: getval  r1,r0           ;get actual proc. type flags
        iori    r0,PRCFLG
;
; get number of locals, args, lines in procedure

        movp    <#,NLOCS>,r1             ;r1 points to word to store
                                         ;number
STUFF1: call    NEXT,STFERR                     ;get number in SBOOBP
        cvtsi   <@,2>,<@,1>,STFERR
        jmpep   r1,<#,NLINES>,STUFF2
        incp    r1
        jmp     STUFF1
;
;allocate vector for procedure directory

STUFF2: movi    <#,4>,r1                 ;r1 will be size of vector
        addi    NLINES,r1
        movi    r1,NINTS
        addi    NARGS,r1
        addi    NLOCS,r1
        addi    NLINES,r1
        addi    NLINES,r1
        pushp   r2                       ;save r2
        movi    <#,F$VEC>,r2
        movp    <#,PRCDIR>,r3
        host    $ALLOC
        rplvec  <#,NINTS>,PRCDIR,<#,1>,<#,4>     ;store integers in dir.
        popp    r2
;
; store args,locals in directory

        movi    NINTS,r1                        ;offset in dir. to location
                                                ;to store next item

        movi    NARGS,r3
        addi    NLOCS,r3
STUFF3: jmpzi   r3,STUFF4
        inci    r1
        call    NEXT,STFERR
        enter   NT,<@,2>,TMPPTR,STUF3A ;get or create name table entry
        setval  <#,UNDEF>,TMPPTR        ;give new vars. UNDEFined value

STUF3A: getflg  TMPPTR,r0
        tstb    <#,F$NWRT>,r0,STFER2    ;var. is no write
        tstb    <#,EV$PRV>,<EVFLGS,4>,STUF3B
        tstb    <#,F$PRIV>,r0,STFER2    ;var. is privileged

STUF3B: rplvec  <#,TMPPTR>,PRCDIR,r1    ;store in directory
        deci    r3
        jmp     STUFF3
;
;store line no.s & source code in directory (& null parse vectors)
;(line no.s must be sorted in ascending order)

STUFF4: movi    NARGS,r3
        addi    NLOCS,r3
        addi    NINTS,r3
        inci    r3                      ;r3 is offset to parse vector
        movi    <#,5>,r1                ; r1 is offset to first line no.
        movp    <#,NULL>,TMPPTR         ;NULL parse code
        movi    NLINES,LINCNT

STUFF5: jmpzi   LINCNT,STUFF6
        call    NEXT,STFERR             ;get line no.
        cvtsi   <@,2>,TMPINT,STFERR
        rplvec  <#,TMPINT>,PRCDIR,r1    ;store line no.
        rplvec  <#,TMPPTR>,PRCDIR,r3    ;(TMPPTR contains NULL)
        movi    r3,r0
        addi    NLINES,r0               ;r0 is offset to source code
        call    NEXT,STFERR             ;get source code list
        rplvec  r2,PRCDIR,r0            ;store source code in directory
        inci    r3
        inci    r1
        deci    LINCNT
        jmp     STUFF5
;
;make procedure directory value of PT entry

STUFF6: jmpnzi  ARGCNT,STFERR           ;all args. should be exhausted
        gettyp  PRCDIR,r0
        iori    PDFLAG,r0               ;set pre-existing proc. dir. flags
        settyp  r0,PRCDIR
        setval  PRCDIR,PRCENT           ;overwrite old procedure dir.
                                        ;if it existed
        iori    NARGS,PRCFLG            ;store number of args in flag part
        setflg  PRCFLG,PRCENT
        pop     r3
        jmp     CMDDON                  ;std. command return

STFERR: movi    <#,E$IMPF>,ERRCOD       ;improper format
        pop     r3
        jmp     STDERR

STFER2: movi    <#,E$OBPV>,ERRCOD
        pop     r3
        jmp     STDERR

;
; TEXTT utility routines--drive buffer management through LGOSTR's string
; buffers & concatenation routines

        extrn   SBIOBP, SBOOBP, SBOBUF, SBOFLG  ;string BCB entries
        extrn   CATLST,CONCAT           ;concatenation routines
        extrn   BOUINI,BOUP             ;buffer mgr. routines
;[
;routine:TXTINI -- sets up LGONAM globals for creating procedure TEXTT(& TITLEs)
;input regs: r2--indirect ptr. to procedure name (a la primitives)
;output: sets up LGONAM temp. vars.
;attr:SAFE
;type: called by TEXTT, TITLE
;globals: SBOBUF,SBOFLG
;]

TXTINI: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,TXTFAI    ;proc. name must be word.
        find    PT,<@,2>,PRCENT                 ;look up proc. table entry
        jmpep   <#,NULL>,PRCENT,TXTFAI  ;jump if not there
        getflg  PRCENT,PRCFLG
        movi    <#,0>,NLOCS             ;defaults for primitives
        movi    PRCFLG,NARGS
        andi    <#,F$ARGS>,NARGS

        tstb    <#,F$PRIM>,PRCFLG,TXTIN2;don't get proc. dir. for prims.
        getval  PRCENT,r0               ;get proc. dir.
        jmpep   r0,<#,UNDEF>,TXTFAI     ;proc. has been erased
        subvec  r0,<#,NINTS>,<#,1>,<#,4>
;
; init. output BCB

TXTIN2: swapp   <@,2>,PRCNAM
        swapp   <#,EMPTY>,SBOOBP
        movp    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI
        succeed

TXTFAI: fail
;[
;routine: GETPTR —supplies ptr. in TMPPTR to buffer management
;globals: TMPPTR
;attr: SAFE
;]

GETPTR: movp    TMPPTR,r0
        return

;[
;routine: GETOFF —supplies ptr. in proc. dir at given offset
;input: TMPINT—offset in proc. dir. containing ptr.
;output reg: r0
;attr: SAFE
;]

GETOFF: getval  PRCENT,r0               ;get transient ptr. to proc. dir.
        subvec  r0,<#,TMPPTR>,TMPINT    ;get transient ptr. in dir.
        movp    TMPPTR,r0
        movp    <#,NULL>,TMPPTR         ;clear transient ptr.
        return
;[
;routine: GETENT —supplies name field of symbo table entry at given
;               offset in proc. dir
;output reg: r0
;attr: SAFE
;]

GETENT: call    GETOFF                  ;proc. dir. ptr. in r0
        getnam  r0,r0
        return

;[
;routines: TXTLBR,TXTRBR,TXTELE --supplies untouchables to output buffer
;output reg: r0-r2 destroyed
;attr:UNSAFE
;]

TXTLBR: movi    <$,U$LBRK>,r2
        jmp     TXTUNT
TXTRBR: movi    <$,U$RBRK>,r2
        jmp     TXTUNT
TXTELE: movi    <$,U$SPCS>,r2
TXTUNT: movp    <$,SBOBUF>,r1
        jmp     BOUP                        ;return through BOUP

;[
;routine: TXTTTL --outputs title line to SBOBUF
;output regs: destroys r0-r2
;attr: UNSAFE
;calls: CONCAT, TXTELE,TXTRBR,TXTLBR, BOUP
;globals:
        extrn   TOSTR,TOGSTR,TOQSTR         ;"TO, "TOGREEDY "TOQUOTING
;]

TXTTTL: tstb    <$,F$GRDY>,PRCFLG,TXTTGR
        tstb    <$,F$QUOT>,PRCFLG,TXTTQU
        movp    <$,TOSTR>,TMPPTR
        jmp     TXTTT1

TXTTGR: movp    <$,TOGSTR>,TMPPTR
        jmp     TXTTT1

TXTTQU: movp    <$,TOQSTR>,TMPPTR
TXTTT1: movp    <$,GETPTR>,SBIOBP
        call    CONCAT                      ;output TO-type string
        call    TXTELE
        swapp   PRCNAM,TMPPTR
        call    CONCAT                      ;output procedure name
        swapp   <$,NULL>,TMPPTR             ;dealc. proc. name
        movi    NARGS,r0
        addi    NLOCS,r0
        jmpzi   r0,TXTTDN                   ;if no local vars. then done
        call    TXTELE                      ;output separator
        call    TXTLBR                      ; [
        movi    NINTS,TMPINT                ;offset to proc. vars.
;
; output args,locals
```

```
        movp    <$,GETENT>,SBIOBP         ;give routine to get vars. to buffer

        jmpzi   NARGS,TXTLOC
        movi    NARGS,ARGCNT
TXTARG: tstb    <$,F$PRIM>,PRCFLG,TXTAR1
        jmp     TXTAR2
;
;for primitives, just output * as arg. name

TXTAR1: movp    <$,SBOBUF>,r1
        movi    <$,"*">,r2
        call    BOUP
        jmp     TXTAR3

TXTAR2: inci    TMPINT                    ;offset to next var.
        call    CONCAT                    ;output var.

TXTAR3: deci    ARGCNT
        jmpzi   ARGCNT,TXTLOC
        call    TXTELE
        jmp     TXTARG

TXTLOC: jmpzi   NLOCS,TXTLO2
        movi    NLOCS,ARGCNT
        call    TXTRBR                    ;output ] [
        call    TXTELE
        call    TXTLBR

TXTLO1: inci    TMPINT                    ;offset to next var.
        call    CONCAT                    ;output var.
        deci    ARGCNT
        jmpzi   ARGCNT,TXTLO2
        call    TXTELE
        jmp     TXTLO1

TXTLO2: call    TXTRBR
TXTTDN: return

;[
;routine:TEXTT --returns TEXT of a procedure
;input reg: std. prim. (proc. name in <@,2>)
;output reg: r0
;calls: TXTINI,TXTELE,TXTTTL,TXTLBR,TXTRBR,CATLST,CONCAT
;globals: proc. dir. variables, SBOBUF BCB, SBIOBP
;attr: UNSAFE
;]
        entry   TEXTT
```

```
;
; initialize LGONAM vars., SBOBUF

TEXTT:  call    TXTINI,REMPTY              ;return empty if proc. doesn't exist
;
;output title line to SBOBUF

        call    TXTLBR                     ;output internal [
        call    TXTTTL                     ;output title itself
        call    TXTRBR                     ;output internal ]
;
; set up for outputting procedure source code

        tstb    <#,F$PRIM>,PRCFLG,TEXDON ;prims. have only title line
        getval  PRCENT,r0                  ;get proc. dir.
        gettyp  r0,r0
        andi    <#,F$TEXT>,r0              ;masks flag bits indicating no text
        jmpnzi  r0,TEXDON

        movi    NLINES,r2
        jmpzi   r2,TEXDON
        movi    r2,LINCNT
        addi    NINTS,r2
        addi    NARGS,r2
        addi    NLOCS,r2                   ;r2 is offset to source lines
                                           ;in proc. dir.
        movi    r2,SRCOFF
        movi    <#,4>,LINOFF               ;LINOFF is offset to line numbers
TEXLIN: inci    SRCOFF
        inci    LINOFF
        call    TXTELE                     ;output separator
        call    TXTLBR
        getval  PRCENT,PRCDIR              ;get transient ptr. to proc. dir.
        subvec  PRCDIR,<#,TMPINT>,LINOFF        ;get line number
        movp    <#,NULL>,PRCDIR            ;clear PRCDIR
        cvtis   TMPINT,TMPPTR
        movp    <#,GETPTR>,SBIOBP          ;give CONCAT routine for buffer mgr.
        call    CONCAT                     ;output line no. string in TMPPTR
        swapp   <#,NULL>,TMPPTR            ;dealc. string
        call    TXTELE
        movp    <#,GETOFF>,SBIOBP          ;give CONCAT routine for buffer mgr.
        movi    SRCOFF,TMPINT
        call    CONCAT
        call    TXTRBR
        deci    LINCNT
        jmppi   LINCNT,TEXLIN

TEXDON: jmp     CATLST                     ;return list to caller
```

```
;
;       Workspace listing routines
;[
;routine: GETNM -- returns name field of table entry referenced by PRCENT
;               for buffer mgr.
;outputs:r0
;attr:SAFE
;]

GETNM:  getnam  PRCENT,r0
        return

;[
;routines: PROCS,NAMES,IPROCS,INAMES,PPROCS,PNAMES
;         returns a list of the appropriate names from a name table.
;regs: argumentless primitives
;attr: UNSAFE
;calls: CONCAT,CATLST, buffer mgr.
;globals: SBOBUF,SBIBUF bcbs,PRVUSR
        extrn   PT,NT
;]
        vsect

SPCFLG: int     0                       ;non-zero if list separator should
                                        ; be output
NAMCHK: ptr     NULL                    ;addr. of routine to filter out
                                        ; unwanted names
HTABLE: ptr     NULL                    ;ptr. to hash table

        psect

        entry PROCS,NAMES,IPROCS,INAMES,PPROCS,PNAMES
;
;all the name checking routines are called with the name table entry
; in PRCENT, the value of the entry in R3, and the entry's flags in R2.
; They fail if the name is to be rejected

;
; PROCS--all non-primitive,non-invisible procedure names

PROCS:  movp    PT,HTABLE
        movp    <#,PROC1>,NAMCHK
        jmp     NAMLST
;
;PROCS name checking

PROC1:  tstb    <#,F$PRIM>,r2,NCHKFL
```

```
        jmpnzi  PRVUSR,PROC2
        tstb    <#,F$PRIV>,r2,NCHKFL
PROC2:  gettyp  r3,r0                   ;get proc. directory flags
        tstb    <#,F$PINV>,r0,NCHKFL
;
;all succeed jumps in name checking come here

NCHKSC: succeed
;
;IPROCS--all invisible, non-primitive procedures

IPROCS: movp    PT,HTABLE
        movp    <#,IPROC1>,NAMCHK
        jmp     NAMLST
;
;IPROCS name checking

IPROC1: tstb    <#,F$PRIM>,r2,NCHKFL
        jmpnzi  PRVUSR,IPROC2
        tstb    <#,F$PRIV>,r2,NCHKFL
IPROC2: gettyp  r3,r0
        tstb    <#,F$PINV>,r0,NCHKSC
;
;all fail jumps in name checking come here

NCHKFL: fail

;
;PPROCS--all non-invisible primitive procedures

PPROCS: movp    PT,HTABLE
        movp    <#,PPROC1>,NAMCHK
        jmp     NAMLST
;
;PPROCS name checking

PPROC1: tstb    <#,F$PRIM>,r2,PPROC2
        jmpnzi  PRVUSR,NCHKFL           ;for priv. users,F$PRIM flag must be on
        tstb    <#,F$PRIV>,r2,PROC2     ;jump to PROCS to finish checking
        fail
PPROC2: jmpnzi  PRVUSR,NCHKSC
        tstb    <#,F$PRIV>,r2,NCHKFL    ;invisible primitive
        succeed
;
;NAMES--all non-privileged,non-invisible NAMES

NAMES:  movp    NT,HTABLE
        movp    <#,NAME1>,NAMCHK
        jmp     NAMLST
;
;NAMES name checking

NAME1:  tstb    <#,F$NINV>,r2,NCHKFL
NAME2:  jmpnzi  PRVUSR,NCHKSC
        tstb    <#,F$PRIV>,r2,NCHKFL
        succeed
;
;INAMES--all non-privileged,invisible names

INAMES: movp    NT,HTABLE
        movp    <#,INAME1>,NAMCHK
        jmp     NAMLST
;
;INAMES name checking

INAME1: tstb    <#,F$NINV>,r2,NCHKSC
        jmp     NAME2
;
;PNAMES--all privileged, non-invisible names

PNAMES: movp    NT,HTABLE
        movp    <#,PNAME1>,NAMCHK
        jmp     NAMLST
;
;PNAMES name checking

PNAME1: tstb    <#,F$NINV>,r2,NCHKFL
        tstb    <#,F$PRIV>,r2,NCHKSC
        fail
;
; the real work starts here

NAMLST: push    r3
        subvec  HTABLE,<#,TMPINT>,<#,1> ;length of hash table
;
; init. output BCB

        movp    <#,EMPTY>,SBOOBP
        movp    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI

        movp    <#,GETNM>,SBIOBP        ;routine to get name field ptr.
        movi    <#,0>,SPCFLG
;
;loop thru hash table from bottom up
```

```
NAMLP:  subvec  HTABLE,<#,PRCENT>,TMPINT        ;ptr. to first table entry in
                                        ; linked hash list
NAMLP1: jmpep   <#,NULL>,PRCENT,NAMLP2  ;jump if no more entries with this
                                        ; hash code
;test if name should be output

        getval  PRCENT,r3
        jmpep   <#,UNDEF>,r3,NAMLP4     ;no value, don't output

        getflg  PRCENT,r2
        movp    NAMCHK,r0               ;get addr. of checking routine
        call    <@,0>,NAMLP4

        jmpzi   SPCFLG,NAMLP3           ;check if list separator should
        call    TXTELE                  ; be output

NAMLP3: call    CONCAT                  ;else add on name
        inci    SPCFLG                  ;& set flag

NAMLP4: getnxt  PRCENT                  ;advance to next entry in list
        jmp     NAMLP1

NAMLP2: deci    TMPINT
        jmpgi   TMPINT,<#,1>,NAMLP      ;advance to next item in hash table

        pop     r3
        jmp     CATLST                  ;return through CATLST
;
;NOTEXT
;
;[
;ROUTINE: NTXRCH --supplies characters to parser
;               fails on end of line
;output REG:r0
;CALLS:
        extrn LEXCVT
;]
        vsect

TMPCHR: int     0                       ;the character returned

        psect

NTXRCH: substr  SRCLIN,<#,TMPCHR>,SRCOFF,<#,1>,NXTRDN
        movi    TMPCHR,r0
        inci    SRCOFF
        jmp     LEXCVT                  ;return thru routine for converting
```

```
                                        ; untouchables

NXTRDN: fail

;[
;ROUTINE:NOTEXT -- removes source code from procedure
;input REGS: std. normal prim.
;ATTR:UNSAFE
;GLOBALS: PT, EMPTY, NLINES, SRCOFF
        extrn LEXRCH
;CALLS:
        extrn   PARSE
;]

        vsect

SLINNO: int     0                       ;offset in proc. dir. to source
PLINNO: int     0                       ; and parsed code
SRCLIN: ptr     NULL                    ;holds text of current line
TMPLIN: ptr     NULL

        psect

        entry   NOTEXT

NOTEXT: push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,NTXER1
        find    PT,<@,2>,r3            ;look up proc. name
        jmpep   <#,NULL>,r3,NTXER1      ;not found
        getflg  r3,r1
        tstb    <#,F$PRIM>,r1,NOTXDN    ;meaningless for primitive
        jmpnzi  PRVUSR,NOTEX1           ;check privileges
        tstb    <#,F$PRIV>,r1,NTXER2

NOTEX1: getval  r3,r0                   ;get proc. dir.
        subvec  r0,<#,NLINES>,<#,4>
        gettyp  r0,r1
        setb    <#,F$BURY>,r1
        settyp  r1,r0                   ;mark as buried
;
;find start of parse & source code

        getlen  r0,r1
        subi    NLINES,r1
        movi    r1,SLINNO
        subi    NLINES,r1
        movi    r1,PLINNO
```

```
;
;loop thru proc. dir., parsing unparsed lines & throwing away source code

        movp    <#,NTXRCH>,LEXRCH       ;give to parser
NOTEX2: jmpzi   NLINES,NOTXDN
        deci    NLINES
        inci    SLINNO
        inci    PLINNO
        subvec  r0,<#,TMPLIN>,PLINNO    ;get parse vector ptr.
        jmpnep  <#,NULL>,TMPLIN,NOTEX3  ;if not null, already parsed
;
;parse line

        subvec  r0,<#,TMPLIN>,SLINNO    ;get transient ptr. to source
        swapp   TMPLIN,SRCLIN           ;give ownership to SRCLIN
        movi    <#,1>,SRCOFF            ;initialize NTXRCH
        call    PARSE,NTXER3
        omovp   r0,TMPLIN               ;trans. ptr. to parse vector
        getval  r3,r0                   ;restore proc. dir. ptr.
        rplvec  <#,TMPLIN>,r0,PLINNO    ;store parse vector
        swapp   <#,NULL>,SRCLIN         ;dealc. old source code
;
;throw way source code (replace with EMPTY)

NOTEX3: movp    <#,EMPTY>,TMPLIN
        rplvec  <#,TMPLIN>,r0,SLINNO
        jmp     NOTEX2

NOTXDN: pop     r3
        jmp     CMDDON
;
;NOTEXT errors

NTXER1: pop     r3
        movi    <#,E$NXOB>,ERRCOD
        jmp     STDERR

NTXER2: pop     r3
        movi    <#,E$OBPV>,ERRCOD
        jmp     STDERR

NTXER3: getval  r3,r0                   ;get proc. dir.
        rplvec  <#,SRCLIN>,r0,SLINNO    ;replace source code that didn´t
        movi    <#,E$TXDP>,ERRCOD       ; parse
        swapp   <#,NULL>,ERROBJ         ;dealc. parser error object
        pop     r3
        jmp     STDERR
```

```
;
;SETPROPERTY supporting routines
;[
;ROUTINE:CHKTYP--checks if inputs to SETPROPERTY
;input REGS: r1--name table, r2--arg. buffer ptr., r3--property type flag
;                                                 (1 of F$PPRP, F$NPRP)
;output REGS: r0--address of entry of object, r1--flags in entry
;               r3--value field of property entry
;ATTR:SAFE
;GLOBALS: ERRCOD,SPT
;]

CHKTYP: push    r1
        find    SPT,<MWRDSZ,2>,r0       ;lookup property
        jmpep   <#,NULL>,r0,CHKER2      ;not found
        getflg  r0,r1                   ;check if property applies here
        tstb    r3,r1,CHKTY2
        jmp     CHKER2

CHKTY2: getval  r0,r3                   ;get property value
        pop     r1
        find    r1,<@,2>,r0             ;lookup name or procedure
        jmpep   <#,NULL>,r0,CHKER1      ;not found
        getflg  r0,r1                   ;get flags
        jmpnzi  PRVUSR,CHKTY3           ;skip type check for priv. users
        tstb    <#,F$PRIM>,r1,CHKER3    ;non-priv. users can't change
        tstb    <#,F$PRIV>,r1,CHKER3    ; effective prims.
CHKTY3: succeed
;
;CHKTYP errors

CHKER1: movi    <#,E$NXOB>,ERRCOD       ;non-existent object
        fail

CHKER2: pop     r1
        decp    r2                      ;r2 points to property
        movi    <#,E$NXSP>,ERRCOD       ;non-existent property
        fail                            ;(or inappropriate)

CHKER3: movi    <#,E$OBPV>,ERRCOD       ;object protection violated
        fail

;[
;ROUTINES:SETPRC,USTPRC,SETNAM,USTNAM
;       (SETPROPERTY#PROCEDURE,UNSETPROPERTY#PROCEDURE,
;        SETPROPERTY#NAME,UNSETPROPERTY#NAME)
; REGS: std. normal prim.
;ATTR:SAFE
;CALLS:CHKTYP, CMDDON, STDERR
;GLOBALS:NT,PT
;]
        vsect

BITFLG: int     0                               ;non-zero if property is being set
                                                ; zero if being unset
        psect
;
;dispatch table for setting procedure properties

STTPTB: ptr     STPPRM
        ptr     STPINV
        ptr     STPTRC
        ptr     STPSTP
        ptr     STPPRD
        ptr     STPPWR
;
;flag table for name properties

STINTB: int     F$PRIV
        int     F$NINV
        int     F$TRCE
        int     0                               ;can't set STEPPED for names
        int     F$NRD
        int     F$NWRT

        entry STTPRC,USTPRC,STINAM,USTNAM

USTPRC: movi    <#,0>,BITFLG
        jmp     STTPR1

STTPRC: movi    <#,1>,BITFLG
STTPR1: push    r3
        movi    <#,F$PPRP>,r3
        movp    PT,r1
        call    CHKTYP,STTERR                   ;check arg. to SET or UNSET PROP.
        movp    <STTPTB,3>,r3                   ;get routine addr. from dispatch table
        jmp     <@,3>


USTNAM: movi    <#,0>,BITFLG
        jmp     STINM1

STINAM: movi    <#,1>,BITFLG
STINM1: push    r3
        movi    <#,F$NPRP>,r3
        movp    NT,r1
        call    CHKTYP,STTERR           ;check arg. to SET or UNSET NAME
        movi    <STNTB,3>,r3            ;get flag bit to set/unset
        jmp     STTTAB                  ; in name table
;
;property setting routines
;all routines will set up a flag word in r3 (leaving the symbol table entry
; in r0, and the flags in r1) and jump to STTPPD to set the prcedure directory
; flags, or to STTTAB to set the symbol table entry flags
;
;
;set procedure PRIMITIVE

STPPRM: tstb    <#,F$PRIM>,r1,STTPM1    ;if LIL prim., error if unsetting else
                                        ; no-op
        movi    <#,F$PRIV>,r3
        jmp     STTTAB
;
;come here for things which are no-ops if set in LIL primitives and errors
; if unset in LIL primitives

STTPM1: jmpzi   BITFLG,STTER1
        jmp     STTDON
;
;set procedure INVISIBLE

STPINV: movi    <#,F$PRIV>,r3           ;for LIL prims., set PRIV flag
        tstb    <#,F$PRIM>,r1,STTTAB
        movi    <#,F$PINV>,r3           ;else set PINV flag in directory
        jmp     STTPD
;
;set TRACE for procedures

STPTRC: movi    <#,F$TRCE>,r3
        jmp     STTTAB
;
;set procedure STEPPED

STPSTP: tstb    <#,F$PRIM>,r1,STTER2    ;non-existent property
        movi    <#,F$STEP>,r3
        jmp     STTPD
;
;set procedure NOREAD

STPPRD: tstb    <#,F$PRIM>,r1,STTPM1    ;can't set for prims
        movi    <#,F$PRD>,r3
        jmp     STTPD
;
;set procedure NOCHANGE
```

```
STPPWR: tstb    <#,F$PRIM>,r1,STTPM1      ;can't set for prims
        movi    <#,F$PWRT>,r3
        jmp     STTPD
;
;do actual set/unset in symbol table
;(r0 has entry address, r1 has current flag word, r3 has bit in question)

STTTAB: jmpzi   BITFLG,USTTAB             ;jump if unstt
        setb    r3,r1
        jmp     STTTA2
USTTAB: clrb    r3,r1
STTTA2: setflg  r1,r0
STTDON: pop     r3
        jmp     CMDDON
;
;do actual set/unset in procedure directory

STTPD:  getval  r0,r0                     ;get proc. dir.
        gettyp  r0,r1                     ;get flags
        jmpzi   BITFLG,USTPD
        setb    r3,r1
        jmp     STTPD2
USTPD:  clrb    r3,r1
STTPD2: settyp  r1,r0
        pop     r3
        jmp     CMDDON
;
;[
;ROUTINES:PRPPRC,PRPNAM (PROPERTY?#PROCEDURE,PROPERTY?#NAME)
;REGS:std. normal prim.
;CALLS: CHKTYP
;ATTR:SAFE
;GLOBALS: PT,NT
        extrn FALSP, TRUEP
;]
;
;dispatch table for testing procedure properties

PRPPTB: ptr     PRPPRM
        ptr     PRPINV
        ptr     PRPTRC
        ptr     PRPSTP
        ptr     PRPPRD
        ptr     PRPPWR
;
;(use STINTB for getting flag bits for name properties)
```

```
        entry   PRPPRC,PRPNAM

PRPPRC: push    r3
        movi    <#,F$PPRP>,r3
        movp    PT,r1
        call    CHKTYP,STTERR           ;check args.
        movp    <PRPPTB,3>,r3           ;get routine addr.
        jmp     <@,3>
;
;procedure property testing routines

        const   F$PPRV,F$PRIM+F$PRIV

PRPPRM: andi    <#,F$PPRV>,r1
        jmpnzi  r1,PRPTRU
        jmp     PRPFLS

PRPINV: movi    <#,F$PRIV>,r3
        tstb    <#,F$PRIM>,r1,PRPTAB    ;test in table entry
        movi    <#,F$PINV>,r3
        jmp     PRPPD                   ;test in proc. dir.

PRPTRC: tstb    <#,F$TRCE>,r1,PRPTRU
        jmp     PRPFLS

PRPSTP: tstb    <#,F$PRIM>,r1,PRPFLS
        movi    <#,F$STEP>,r3
        jmp     PRPPD

PRPPRD: tstb    <#,F$PRIM>,r1,PRPTRU
        movi    <#,F$PRD>,r1
        jmp     PRPPD

PRPPWR: tstb    <#,F$PRIM>,r1,PRPTRU
        movi    <#,F$PWRT>,r1
        jmp     PRPPD
;
;test if bit is on in proc. dir

PRPPD:  getval  r0,r0
        gettyp  r0,r1
        tstb    r3,r1,PRPTRU
        jmp     PRPFLS
;
;
;
PRPNAM: push    r3
        movi    <#,F$NPRP>,r3
        movp    NT,r1
        call    CHKTYP,STTERR
        movi    <STINTB,3>,r3           ;get bit to check
;
;test if bit is on in table entry

PRPTAB: tstb    r3,r1,PRPTRU
;
;return false

PRPFLS: movp    <#,FALSP>,r0
        pop     r3
        succeed
;
;return true

PRPTRU: movp    <#,TRUEP>,r0
        pop     r3
        succeed
;
;errors in property settting, testing

STTER1: movi    <#,E$OBPV>,ERRCOD       ;object prot. violation
        jmp     STTERR

STTER2: decp    r2
        movi    <#,E$NXSP>,ERRCOD
STTERR: pop     r3
        jmp     STDERR

        endmod LGONAM
```

4.2.12 LGOSTR.LIL

```
       search LIL

        module LGOSTR — LOGO string handling primitives

        include FLAGS
        include CONSTS
        include ERRORS
;
        vsect
```

```
; temporary locations for storing the standard args. to a primitive
; (rl: # of args.; r2: ptr to arg. buffer)

ARGCNT: int     0
ARGPTR: ptr     null

;temporary storage

NUMLEN: int 0
TMPINT: int     0
TMPPTR: ptr     null

;       STRING buffers --available to all primitives

; input buffer

        entry SBISIZ,SBICNT,SBIPTR,SBIOBP,SBIOBC,SBIOBL,SBIFLG,SBIBUF

SBISIZ: int    SBUFSZ
SBICNT: int    0
SBIPTR: ptr    SBIBUF
SBIOBP: ptr    null
SBIOBC: int    0
SBIOBL: int    0
SBIFLG: int    B$STR
SBIBUF: blki   SBUFSZ

; output buffer


        entry SBOSIZ,SBOCNT,SBOPTR,SBOOBP,SBOOBC,SBOOBL,SBOFLG,SBOBUF

SBOSIZ: int    SBUFSZ
SBOCNT: int    0
SBOPTR: ptr    SBOBUF
SBOOBP: ptr    null
SBOOBC: int    0
SBOOBL: int    0
SBOFLG: int    B$STR
SBOBUF: blki   SBUFSZ

; Buffer management routines

        extrn   BININI,BINP,BOUINI,BOUP,BOUEND,SETINP,GETBUF
        extrn   RBOUP,RBINP,RGTBUF,RPTBUF,RBOINI
        psect
```

```
;
        subttl Standard error failure
;
; The failing routine sets up ERRCOD
; r2 is assumed to point to the ERROBJ
;
; global error words

        extrn   ERRCOD                  ; Error code (integer)
        extrn   ERRINF                  ; error contxt
        extrn   ERROBJ                  ; indirect ptr. to offending Argument

        entry   STDERR

STDERR: movp    r2,ERROBJ
        fail

;
;       Standard return empty for all primitives
;
        entry   REMPTY

        extrn   EMPTY

REMPTY: movp    <#,EMPTY>,r0
        succeed

        entry   RZERO

;
;   return zero

        extrn   CRNUM

RZERO:  movi    <#,1>,r1
        call    CRNUM
        succeed

;
;standard return for commands

        entry   CMDDON

CMDDON: movp    <#,UNDEF>,r0
        succeed
;
;       Concatenation routines --perform concatentation of args. for
;        LGOSTR & LGONAM primitives
```

```
;[
;routine: CATINI --initializes BCB's for concatentation
;input regs: r1-# of args.,r2-arg.buffer ptr.
;attr: UNSAFE
;globals: STRING BCB's, ARGCNT,ARGPTR
;calls:BOUINI
;attr: SAFE
;]
        entry   CATINI

CATINI: movi    r1,ARGCNT
        movp    r2,ARGPTR
        movp    <#,GETARG>,SBIOBP       ;give addr. of routine to get obj.
                                        ; for buffer management
        swapp   <#,EMPTY>,SBOOBP                ;set up output object
        movi    <#,B$STR>,SBOFLG        ;output will be string
        movp    <#,SBOBUF>,r1
        call    BOUINI                  ;init. output BCB
        return

;[
;routine: CONCAT -- concatenates current arg. onto object referenced
;                   by output BCB (contents of SBOOBP)
;globals: string BCB, ARGCNT,ARGPTR
;calls: BININI,BINP,BOUP,BOUEND
;attr: UNSAFE
;temp. regs: r0,r1,r2 destroyed
;]
        entry   CONCAT

; set up arg. for input; init. input BCB
;
CONCAT: movi    <#,B$STR>,SBIFLG
        movi    <#,1>,SBIOBC            ;offset to start of object
        movp    <#,SBIBUF>,r1           ;address of input BCB
        call    BININI                  ;initialize input BCB
;
; inner loop--copy arg. to output object
;
CONCA2: call    BINP                    ;get input char. in r0
        movi    r0,r2
        tstb    <#,B$DONE>,SBIFLG,CONCA3 ;jump if input object finished
        movp    <#,SBOBUF>,r1
        call    BOUP                    ;put it in output object
        movp    <#,SBIBUF>,r1
        jmp     CONCA2                  ; else loop back
```

```
CONCA3: deci    ARGCNT
        decp    ARGPTR
        return

;
;CATLST -- standard successful return for primitives which use
;       concatentation & return a list

        entry   CATLST

CATLST: movp    <#,SBOBUF>,r1
        call    BOUEND                  ; close output buffer
        omovp   SBOOBP,r0               ; get output object
        jmpep   r0,<#,EMPTY>,CTLDON
        gettyp  r0,r1                   ; get flags
        setb    <#,F$LIST>,r1
        settyp  r1,r0                   ; mark as list
CTLDON: succeed

;
;       string utility routines

;
;[
;routine: GETARG --gives addr. of next input object to buffer management
;called by: GETBUF,BININI
;output reg: r0
;global: ARGPTR
;attr: SAFE
;]

GETARG: movp    ARGPTR,r0
        movp    <@,0>,r0
        return

;[
;routine: NXTLST -- moves thru a sub-list
;input:          Uses the string buffer SBIBUF as its primary input.
;               The BCB of SBIBUF is assumed to refer to a list and to
;               be pointing to the first byte of an sub-list.
;
;input reg: r3 -- If r3 is non-zero, NXTLST will output the sub-list
;               using the string buffer SBOBUF which is assumed to be
;               properly initialized.
;                 If r3 > 1, NXTLST does not output the 1st left
;               bracket of the sub-list.(The closing right bracket
;               is never output)
;
;output:          If r3 is non-zero, the output object (the sub-list)
;                will be in SBOOBP when SBOBUF is closed (via BOUEND).
;       r0 returns greatest depth of sub-list
;       r1,r2 destroyed
;attr: UNSAFE
;type: string utility
;]

        vsect

LSTLEV: int     0                       ; current depth of list
MAXLEV: int     0                       ; greatest depth of list

        psect

        entry   NXTLST

NXTLST: movi    <#,0>,LSTLEV            ; initial list depth is 0
        movi    <#,0>,MAXLEV

NXTLP:  movp    <#,SBIBUF>,r1
        call    BINP                    ; get next char. in list
        tstb    <#,B$DONE>,SBIFLG,NXDONE ;  object is finished, list done
        jmpei   r0,<#,U$SPCS>,LSTDON    ; end of sub-list char. found
        jmpei   r0,<#,U$LBRK>,LSTPSH    ; start of sub-list
        jmpei   r0,<#,U$RBRK>,LSTPOP    ; end of sub-list
;
; if r3 is non-zero, output char.

LSTOUT: jmpzi   r3,NXTLP
LSTOU2: movp    <#,SBOBUF>,r1           ; output BCB
        movi    r0,r2                   ; output char.
        call    BOUP
        jmp     NXTLP

LSTPSH: inci    LSTLEV                  ; start of sub-list, increment level
        jmpngi  LSTLEV,MAXLEV,LSTPS2    ; if new level is greater than current
        movi    LSTLEV,MAXLEV           ;  maximum, adjust maximum
LSTPS2: jmpgi   LSTLEV,<#,1>,LSTOUT
        jmpei   r3,<#,1>,LSTOU2         ; if 1st left bracket,only output
        jmp     NXTLP                   ; if r3 = 1

LSTPOP: deci    LSTLEV                  ; end of sub-list, decr. level
        jmpnzi  LSTLEV,LSTOUT
        movp    <#,SBIBUF>,r1
        call    BINP                    ; move past closing bracket
        jmp     NXDONE
```

```
LSTDON: jmpnzi  LSTLEV,LSTOUT           ; if top level of list, then end
                                        ;  of sub-list found,else keep going
NXDONE: movi    MAXLEV,r0               ; return maximum depth of sub-list
        return

;
;[
;routine: PREVLS --moves thru last element of current arg. in an
;                analogous manner to NXTLST.  Initializes input, output
;                buffers.
;input regs: r2 --indirect ptr. to object
;            r3 -- if non-zero copy object without outermost brackets
;attr: UNSAFE
;calls: reverse buffer management
;]

PREVLS: movp    r2,ARGPTR               ;so GETARG will work
        movp    <@,2>,r2               ;r2 is direct pointer
        movp    <#,GETARG>,SBIOBP
        getlen  r2,SBIOBL
        movi    SBIOBL,SBIOBC           ;this starts reading from end
                                        ;of object
        movi    <#,B$STR>,SBIFLG
        movp    <#,SBIBUF>,r1
        call    RGTBUF                  ;load in first buffer

        jmp i   r3,PREVL2               ;check if output buffer should
                                        ; be opened
;
; open output buffer

        swapp   <#,EMPTY>,SBOOBP
        movp    <#,SBOBUF>,r1
        call    RBOINI                  ;init. BUFCNT,BUFPTR
        movi    <#,B$STR>,SBOFLG
        movi    <#,0>,SBOOBL            ;zero length initially

PREVL2: movi    <#,0>,LSTLEV
        movi    <#,0>,MAXLEV            ;from NXTLST

PREVLP: movp    <#,SBIBUF>,r1
        call    RBINP                   ; get previous char. in list
        tstb    <#,B$DONE>,SBIFLG,PRDONE ;  object is finished, list done
        jmpei   r0,<#,U$SPCS>,PRVDON    ; end of sub-list char. found
        jmpei   r0,<#,U$RBRK>,PRVPSH    ; start of sub-list
        jmpei   r0,<#,U$LBRK>,PRVPOP    ; end of sub-list
;
; if r3 is non-zero, output char.
```

```
PRVOUT: jmpzi   r3,PREVLP
        movp    <#,SBOBUF>,r1           ; output BCB
        movi    r0,r2                   ; output char.
        call    RBOUP
        jmp     PREVLP

PRVPSH: inci    LSTLEV                  ; start of sub-list, increment level
        jmpngi  LSTLEV,MAXLEV,PREVP2    ; if new level is greater than current
        movi    LSTLEV,MAXLEV           ;  maximum, adjust maximum
PREVP2: jmpgi   LSTLEV,<#,1>,PRVOUT     ;dont print outermost bracket
        jmp     PREVLP

PRVPOP: deci    LSTLEV                  ; end of sub-list, decr. level
        jmpnzi  LSTLEV,PRVOUT
        movp    <#,SBIBUF>,r1
        call    RBINP                   ; move past closing bracket
        jmp     PRDONE

PRVDON: jmpnzi  LSTLEV,PRVOUT           ; if top level of list, then end
                                        ;  of sub-list found,else keep going
PRDONE: movi    MAXLEV,r0               ; return maximum depth of sub-list
        movp    ARGPTR,r2               ;restore r2
        return

;[
;routine: COPSUB — copies a substring.
;input regs: r2 — starting offset. r3—ending offset.  The input object
;               is assumed to be accessible via the SBIBUF input BCB (which
;               should have been initialized).  The output buffer SBOBUF is
;               set up by COPSUB.
;output reg: r0 — copied substring
;attr: UNSAFE
;calls: BUFFER MANAGEMENT
;type: string utility
;]
        entry COPSUB

;
; the input offsets are checked for legality & adjusted if necessary

COPSUB: jmppi   r2,COPSU1               ;starting offset must be > 0
        movi    <#,1>,r2

COPSU1: jmpngi  r3,SBIOBL,COPSU2        ;ending offset must be <= object
        movi    SBIOBL,r3               ; length

;
```

```
;set up input,output BCB's

COPSU2: subi    r2,r3
        inci    r3                          ;r3 is # of bytes to read
        movp    <#,SBIBUF>,r1
        call    SETINP                      ;reset SBIBUF to starting offset
        swapp   <#,EMPTY>,SBOOBP                    ;initial output object
        movi    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI
;
; main copy loop

COPSLP: jmpnpi  r3,COPDON
        movp    <#,SBIBUF>,r1
        call    BINP
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        deci    r3
        jmp     COPSLP
;
; close output buffer, return object

COPDON: movp    <#,SBOBUF>,r1
        call    BOUEND
        swapp   SBOOBP,r0
        return

;
        entry   WORD

;[
;routine: WORD (both LONGWORD & WORD )
;regs: standard primitive
;output reg: r0
;attr: UNSAFE
;globals : ERRCOD,ERRINF,ERROBJ,ARGCNT,ARGPTR
;calls: CATINI,CONCAT,REMPTY,BOUEND
;]

WORD:   jmpzi   r1,REMPTY                   ;return empty if no args.
        call    CATINI                      ; ARGCNT,ARGPTR, output BCB setup

WORD2:  movp    ARGPTR,r2
        jmpep   <#,EMPTY>,<@,2>,WORD3  ;treat EMPTY as a word
        gettyp  <@,2>,r0                    ;get arg. flags
        tstb    <#,F$LIST>,r0,WLIST         ;can't be a LIST
```

```
WORD3:  call    CONCAT
        jmppi   ARGCNT,WORD2

        movp    <#,SBOBUF>,r1
        call    BOUEND                          ; close output BCB
        omovp   SBOOBP,r0                       ;return output object

        succeed
;
; List found as arg. to WORD
;
WLIST:  movi    <#,E$WELF>,ERRCOD
        swapp   <#,NULL>,SBOOBP                 ;delete partially created output
        jmp     STDERR                          ; standard failure
;
;[
;routine: SENTEN (SENTENCE,LONGSENTENCE)
;input,output regs: standard primitive
;attr: UNSAFE
;calls: CATINI,CONCAT,BOUP,BOUEND
;globals: ARGPTR, ARGCNT
;]
        entry   SENTEN

SENTEN: jmpzi   r1,REMPTY                       ; special no args. case
        push    r3
        call    CATINI                          ; initialize BCB's, ARGCNT,ARGPTR
        movi    <#,0>,r3                        ; r3 is zero if nothing has yet been
                                                ; output
SENTE2: movp    ARGPTR,r2
        jmpep   <#,EMPTY>,<@,2>,SENEMP ;ignore empty args.
        jmpzi   r3,SENTE3                       ;skip if no list sep. to be output
        movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2                   ; if not last arg., insert separator
        call    BOUP                            ; char. between elements of output list
SENTE3: call    CONCAT
        movi    <#,1>,r3
SENTE4: jmppi   ARGCNT,SENTE2                   ;loop if more args.
;
; return with list

        pop     r3
        jmp     CATLST

SENEMP: deci    ARGCNT
        decp    ARGPTR
        jmp     SENTE4
```

```
;
;[
;routine: LIST (LIST, LONGLIST)
;input regs: standard primitive
;output reg: r0
;attr: UNSAFE
;calls: CATINI,CONCAT,BOUP,BOUEND,REMPTY
;globals: ARGPTR,ARGCNT
;called explictly by EVALUATOR
;]
        entry   LIST

LIST:   jmpzi   r1,REMPTY                  ; return EMPTY object if no args.
        call    CATINI                     ; initialize ARGPTR,ARGCNT, output BCB
;
;main loop
;
LIST2:  movp    ARGPTR,r2
        gettyp  <@,2>,r0                  ; if arg. is list, add untouchable
        push    r0                         ; save arg. flags
        tstb    <#,F$LIST>,r0,LIST3        ;  list markers
        jmp     LIST4

LIST3:  movi    <#,SBOBUF>,r1              ; output BCB
        movi    <#,U$LBRK>,r2              ; add start of list marker
        call    BOUP

LIST4:  call    CONCAT
        pop     r0
        tstb    <#,F$LIST>,r0,LIST5        ; if list,
        jmp     LIST6
LIST5:  movi    <#,U$RBRK>,r2              ; add end of list marker
        movp    <#,SBOBUF>,r1
        call    BOUP

LIST6:  jmpzi   ARGCNT,CATLST              ; if last arg. ,return a list to
        movp    <#,SBOBUF>,r1              ;  EVAL, else
        movi    <#,U$SPCS>,r2              ;  add separator character
        call    BOUP
        jmp     LIST2                      ; and loop back
;
;[
;routine: ERRARG (creates a list out of args & procedure name of error call)
;input regs:    r1- # of args.
;               r2- ptr. to arg. buffer
;               r3-ptr. to procedure name
;output reg:    r0
;attr: UNSAFE
;CALLS:
        extrn CVTSTR,GIVEUP
;called by: Evaluator error handlers
;]
        vsect

PRCNAM: ptr     NULL

        psect

GETPRC: movp    PRCNAM,r0               ;feed proc. name ptr. to buffer mgr.
        return

        entry   ERRARG

ERRARG: omovp   r3,PRCNAM
        call    CVTSTR,GIVEUP           ;convert arg. buffer to strings
        call    CATINI
;
; copy procedure name

        movp    <#,GETPRC>,SBIOBP
        call    CONCAT
        inci    ARGCNT                  ;adjust decrements made by
        inci    ARGPTR                  ; CONCAT
;
; copy args

        swapp   <#,NULL>,PRCNAM         ;dealc. procedure name
        movp    <#,GETARG>,SBIOBP
ERRAR0: jmpzi   ARGCNT,CATLST           ;return to caller with list
;
; insert list separator before args.
; append quote to words, place brackets around lists

        movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2
        call    BOUP
        movi    <#,U$LBRK>,r2           ;for lists
        movp    ARGPTR,r0
        gettyp  <@,0>,r3                ;get object type
        tstb    <#,F$LIST>,r3,ERRAR1
        movi    <#,C$QUOT>,r2           ;for words
ERRAR1: call    BOUP

        call    CONCAT
;
;add closing bracket for lists
```

```
        tstb    <#,F$LIST>,r3,ERRAR2
        jmp     ERRAR0
ERRAR2: movp    <#,SBOBUF>,r1
        movi    <#,U$RBRK>,r2
        call    BOUP
        jmp     ERRAR0

;
;[
;routine: FRSPUT -- (FIRSTPUT). Expects 2 args., 2nd a list
;input regs: standard normal primitive
;output reg: r0
;attr: UNSAFE
;type: primitive
;calls: CATINI,CONCAT,BOUP,BOUEND
;globals: ARGPTR,ERROBJ,ERRCOD
;called explictly by EVALUATOR
;]
        entry   FRSPUT

FRSPUT: push    r3
        call    CATINI                  ; set up ARGPTR, string buffers
        movp    ARGPTR,r0
        gettyp  <@,0>,r3               ; r3 gets flags of 1st arg.
        tstb    <#,F$LIST>,r3,FRSPU1
        jmp     FRSPU2
;
; if 1st arg. a list , add implicit left bracket

FRSPU1: movp    <#,SBOBUF>,r1
        movi    <#,U$LBRK>,r2
        call    BOUP
;
;copy 1st arg.

FRSPU2: call    CONCAT                  ; ARGPTR now points to 2nd arg.
        tstb    <#,F$LIST>,r3,FRSPU3
        jmp     FRSPU4
;
; if 1st arg. a list , add implicit right bracket

FRSPU3: movp    <#,SBOBUF>,r1
        movi    <#,U$RBRK>,r2
        call    BOUP
;
; 2nd arg. must be a list
```

477

```
FRSPU4: movp    ARGPTR,r2
        jmpep   <#,EMPTY>,<@,2>,FRSPU6 ;ignore 2nd arg. of EMPTY
        gettyp  <@,2>,r3              ; get flags
        tstb    <#,F$LIST>,r3,FRSPU5
        pop     r3
        jmp     FRSERR
;
; add sub-list separator, copy 2nd arg.

FRSPU5: movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2
        call    BOUP
        call    CONCAT
FRSPU6: pop     r3
        jmp     CATLST                 ; make output object a list &
                                       ; return it to EVALUATOR

;
; word instead of list error

FRSERR: movi    <#,E$LEWF>,ERRCOD
        swapp   <#,NULL>,SBOOBP        ; dealc. any temp. object
        jmp     STDERR                 ; standard failure


;
;[
;routine: LASPUT -- (LASTPUT).  Expects 2 args, 1st a list.
;input regs: std. normal primitive
;outputreg: r0
;attr: UNSAFE
;calls: CATINI,CONCAT,buffer management output routines
;]
        entry   LASPUT

LASPUT: push    r3
        call    CATINI
        movp    ARGPTR,r0
        jmpep   <#,EMPTY>,<@,0>,LSPU1A ; 1st arg. if EMPTY
        gettyp  <@,0>,r3              ; check that first arg is a list
        tstb    <#,F$LIST>,r3,LSPUT1
        pop     r3
        jmp     FRSERR                 ; if not return error
;
;  output all of the first argument

LSPUT1: call    CONCAT
        movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2          ; insert separator
        call    BOUP                   ;..between list and thing appended
        jmp     LSPU1B
;
;advance ARGPTR for EMPTY first args. while skipping CONCAT

LSPU1A: decp    ARGPTR

LSPU1B: movp    ARGPTR,r2
        gettyp  <@,2>,r3                ; 2nd arg a list?
        tstb    <#,F$LIST>,r3,LSPUT2
        jmp     LSPUT3
;
;   if thing appended is a list, put in implicit left bracket

LSPUT2: movp    <#,SBOBUF>,r1
        movi    <#,U$LBRK>,r2
        call    BOUP
;
;  copy body of thing to be appended

LSPUT3: call    CONCAT
        tstb    <#,F$LIST>,r3,LSPUT4    ; stick on final right brk.
        jmp     LSPDON                  ;...if needed.
;
;
LSPUT4: movp    <#,SBOBUF>,r1
        movi    <#,U$RBRK>,r2
        call    BOUP
;
;  finish up by returning arg.

LSPDON: pop     r3
        jmp     CATLST

;
;[
;routine: BUTFIRST
;input regs: std. normal prim.(r1 ignored)
;output reg: r0
;attr: UNSAFE
;global: EMPTY, ARGPTR, string buffers
;calls: CATINI,BININI,NXTLST,REMPTY
;]
        entry   BUTFIR

BUTFIR: jmpep   <#,EMPTY>,<@,2>,REMPTY ;BUTFIRST of EMPTY is EMPTY
        movi    <#,B$STR>,SBIFLG        ;set up input buffer BCB
        movp    r2,ARGPTR
        movp    <#,GETARG>,SBIOBP       ; char. routine for buffer mgr.
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI                  ;init. input buffer.
;
; test arg, type

        gettyp  <@,2>,TMPINT            ;save flags
        tstb    <#,F$LIST>,TMPINT,BUTFLS
;
; BUTFIRST of word. Skip over first char.,then go copy

        call    BINP
        jmp     BUTFCP
;
; BUTFIRST of list. skip over 1st sub-list, copy rest

BUTFLS: push    r3
        movi    <#,0>,r3                ;signal NXTLST not to copy
        call    NXTLST                  ; sub-list
        pop     r3
;
;  set up output buffer,copy rest of object

BUTFCP: tstb    <#,B$DONE>,SBIFLG,REMPTY; if done, then only 1 element
                                        ;  return with  EMPTY
        swapp   <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI

BUTFLP: movp    <#,SBIBUF>,r1
        call    BINP
        tstb    <#,B$DONE>,SBIFLG,BUTDON
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        jmp     BUTFLP

BUTDON: jmpzi   SBOCNT,REMPTY           ;if no chars. in output object,
                                        ;return empty
        tstb    <#,F$LIST>,TMPINT,CATLST ;return list
        movp    <#,SBOBUF>,r1
        call    BOUEND
        omovp   SBOOBP,r0               ;return word
        succeed

;
;[
```

480

```
;routine: FIRST
;regs: standard normal primitive (rl ignored)
;attr: UNSAFE
;global: empty
;calls: GETARG,buffer management,NXTLST
;]
        entry   FIRST

FIRST:  jmpep   <#,EMPTY>,<@,2>,REMPTY ; FIRST of empty object is empty
        push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,FIRLST
;
; FIRST of a word is first byte

        substr  <@,2>,<#,TMPINT>,<#,1> ; first byte in TMPINT
;
; entry point for CHAR & ELEMENT to return 1 char. string

FIRWRD: movi    <#,1>,r1                ; allocate 1 byte string
        movi    <#,F$STR>,r2
        movp    <#,TMPPTR>,r3           ;  in TMPPTR
        host    $ALLOC
        rplstr  <#,TMPINT>,TMPPTR       ; store byte in string
        omovp   TMPPTR,r0               ; return string
        jmp     FIRDON
;
;FIRST of a list -- use NXTLST to copy first element

FIRLST: call    CATINI                  ; sets up output buffer,ARGPTR

        movi    <#,B$STR>,SBIFLG        ; input buffer will point
        movi    <#,1>,SBIOBC            ;  to start of list
        movp    <#,SBIBUF>,r1
        call    BININI

FIRCPY: movi    <#,2>,r3                ; signal NXTLST to copy list element
        call    NXTLST                  ; (without outermost brackets)
        movi    r0,r3                   ; save maximum depth of list element
        movp    <#,SBOBUF>,r1
        call    BOUEND                  ; close output object
        omovp   SBOOBP,r0               ;  & return word
        jmpzi   r3,FIRDON               ; if maximum depth was > 0,
        jmpep   r0,<#,EMPTY>,FIRDON     ; or object is not empty list,then
        gettyp  r0,r1                   ; mark object as list
        setb    <#,F$LIST>,r1
        settyp  r1,r0
```

```
FIRDON: pop     r3
        succeed
;
;[
;routine: LAST
;regs: std. normal primitive
;attr:UNSAFE
;calls: FIRWRD,reverse buffer management,PREVLS
;]
        entry   LAST

LAST:   jmpep   <#,EMPTY>,<@,2>,REMPTY ; LAST of empty object is empty
        push    r3
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,LASLST
;
;LAST of a word is last byte

        getlen  <@,2>,r0                ;get offset of last byte
        substr  <@,2>,<#,TMPINT>,r0     ; put byte in TMPINT
        jmp     FIRWRD                   ;reutrn 1 byte word
;
;LAST of a list -- use PREVLS to copy last element

LASLST: movi    <#,2>,r3                 ;signal PREVLS to copy object
        call    PREVLS                   ;initializes SBIBUF,SBOBUF as well
                                         ;r0 is non-zero if list read
; close SBOBUF & return object

        movi    r0,r3                    ; save maximum depth of list element
        jmpzi   SBOCNT,LASLS1            ;skip if curent output buffer empty
        movp    <#,SBOBUF>,r1
        call    RPTBUF                   ;otherwise, write out last buffer

LASLS1: omovp   SBOOBP,r0                ;  & return word
        jmpzi   r3,LASDON                ; if maximum depth was > 0,
        jmpep   r0,<#,EMPTY>,FIRDON      ; or object is not empty list,then
        gettyp  r0,r1                    ; mark object as list
        setb    <#,F$LIST>,r1
        settyp  r1,r0

LASDON: pop     r3
        succeed
;
;[
;routine:BUTLAS (BUTLAST)
;regs: std. normal primitive
;attr: UNSAFE
;calls: reverse buffer mgr.
;]
        entry   BUTLAS

BUTLAS: jmpep   <#,EMPTY>,<@,2>,REMPTY
        gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,BTLLST
;
; BUTLAST of word is all but last byte

        call    CATINI                  ;set up output BCB
        movi    <#,B$STR>,SBIFLG
        movp    ARGPTR,r0
        movp    <@,0>,r2               ;direct ptr. to object
        getlen  r2,SBIOBL
        deci    SBIOBL                  ;make buffer mgr. think that
        movi    <#,1>,SBIOBC            ;input object is 1 byte smaller
        movp    <#,SBIBUF>,r1
        call    GETBUF
        call    CONCA2                  ;enter CONCAT after BCB SETUP
        movp    <#,SBOBUF>,r1
        call    BOUEND
        swapp   SBOOBP,r0
        succeed
;
; BUTLAST of list

BTLLST: push    r3
        movi    <#,0>,r3                ;signal PREVLS to skip last element
        call    PREVLS                  ;SBIBUF set up
        pop     r3
;
; now setup SBOBUF and copy rest of object

        swapp   <#,EMPTY>,SBOOBP
        movp    <#,SBOBUF>,r1
        call    RBOINI
        movi    <#,B$STR>,SBOFLG
        movi    <#,0>,SBOOBL

BTLLLP: movp    <#,SBIBUF>,r1
        call    RBINP
        tstb    <#,B$DONE>,SBIFLG,BTLDON
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    RBOUP
        jmp     BTLLLP
;
```

```
;close output buffer & make object list

BTLDON: jmpzi   SBOCNT,BTLDO2           ;write out last buffer if
        movp    <#,SBOBUF>,r1           ;not empty
        call    RPTBUF

BTLDO2: omovp   SBOOBP,r0
        jmpep   <#,EMPTY>,r0,BTLEND
        gettyp  r0,r1
        setb    <#,F$LIST>,r1
        settyp  r1,r0
BTLEND: succeed



;
;[
;routine: ASCIII --returns integer value of 1 - char. word
;regs: std. primitive
;ATTR:unsafe
;CALLS: CRNUM
;]
        entry   ASCIII
;
; input must beword of length 1

ASCIII: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,LSTERR
        getlen  <@,2>,r0
        jmpnei  r0,<#,1>,SIZERR

        substr  <@,2>,<#,TMPINT>       ;get character
        movi    <#,2>,r1
        call    CRNUM                   ;make 2 word number
        rplvec  <#,TMPINT>,r0,<#,2>     ;store character in 2nd word of
        succeed                         ; number

LSTERR: movi    <#,E$WELF>,ERRCOD       ;word expected, list found
        jmp     STDERR

SIZERR: movi    <#,E$WSZO>,ERRCOD       ;wrong size object
        jmp     STDERR

;[
;routine: CHAR --converts a positive integer to a 1 char. word
;regs: std. primitive
;attr: primitive operation
;]
```

```
        entry   CHAR

CHAR:   subvec  <@,2>,<#,NUMLEN>        ;get length of integer
        jmpei   NUMLEN,<#,1>,AORERR      ;zero--error
        jmpgi   NUMLEN,<#,2>,AORERR      ;too big error
        gettyp  <@,2>,r0
        tstb    <#,F$SIGN>,r0,AORERR     ;negative number error
        subvec  <@,2>,<#,TMPINT>,NUMLEN          ;else get integer
;
; jump to part of FIRST which allocates 1 char. string and puts contents of
; TMPINT in it.

        push    r3
        jmp     FIRWRD

NNSERR: movi    <#,E$NENF>,ERRCOD        ;number expected, none found
        jmp     STDERR

;
;[
;routine: COUNT
;input regs: std. primitive
;output reg: r0
;attr: UNSAFE
;calls: BININI, NXTLST, CRNUM
;]

        entry   COUNT

COUNT:  jmpep   <#,EMPTY>,<@,2>,RZERO  ; return 0 if arg is EMPTY
        push    r3
        gettyp  <@,2>,r0                ; are we counting elements of a list?
        tstb    <#,F$LIST>,r0,CNTLST     ; yes... skip ahead
;
;  COUNT of a word simply means getting the length from the prefix

        getlen  <@,2>,TMPINT
        jmp     CNTDON

;
;  COUNT of a list requires the use of NXTLST to go through all the sublists

CNTLST: movi    <#,B$STR>,SBIFLG         ; initialize input BCB
        movp    r2,ARGPTR
        movp    <#,GETARG>,SBIOBP
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI
```

```
;
;  cycle through sublists, counting the elements

        movi    <#,1>,TMPINT            ; initialize counting cell
CNTELM: movi    <#,0>,r3
        call    NXTLST
        tstb    <#,B$DONE>,SBIFLG,CNTDON  ; are we finished with list?
        inci    TMPINT                  ; no, keep counting
        jmp     CNTELM

;  return non-zero number as result

CNTDON: pop     r3
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,TMPINT>,r0,<#,2>
        succeed


;[
;routine ELEMENT
;attr: UNSAFE
;regs: std. primitive
;]

        entry   ELMENT

ELMENT: movi    <#,1>,r1
        call    CVTNU0,ELMERR
        gettyp  <@,2>,r1
        tstb    <#,F$SIGN>,r1,AORERR
        subvec  <@,2>,<#,NUMLEN>
        jmpei   NUMLEN,<#,1>,REMPTY
        jmpgi   NUMLEN,<#,2>,AORERR
        subvec  <@,2>,<#,TMPINT>,<#,2>
        decp    r2
        jmpep   <#,EMPTY>,<@,2>,REMPTY
        movi    <#,1>,r1
        call    CVTSTR,GIVEUP
        gettyp  <@,2>,r1
        tstb    <#,F$LIST>,r1,ELMLST

        getlen  <@,2>,r1
        movi    TMPINT,r0
        jmpgi   r0,r1,REMPTY
        substr  <@,2>,<#,TMPINT>,r0
        push    r3
        jmp     FIRWRD
```

486

```
ELMLST: push    r3
        movi    <#,B$STR>,SBIFLG
        movp    r2,ARGPTR
        movp    <#,GETARG>,SBIOBP
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI

ELMSKP: deci    TMPINT
        jmpzi   TMPINT,ELMCPY
        movi    <#,0>,r3
        call    NXTLST
        tstb    <#,B$DONE>,SBIFLG,ELMLEP
        jmp     ELMSKP

ELMLEP: pop     r3
        jmp     REMPTY

ELMCPY: swapp   <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG
        movp    <#,SBOBUF>,r1
        call    BOUINI
        jmp     FIRCPY

ELMERR: fail

AORERR: movi    <#,E$AOR>,ERRCOD
        jmp     STDERR
;[
;routine: SUBOBJ (SubObject) Takes 3 args, 1st two are numbers and the
;       3rd a word or list.
;input regs: std. primitive
;output reg: r0
;attr: UNSAFE
;calls: NXTLST, buffer management, error regs.,CVTSTR
        extrn CVINU0
;]
        entry   SUBOBJ

SUBOBJ: push    r3
        movi    <#,2>,r1
        call    CVINU0,SBJERR           ;check args 1 & 2 are numbers
        movi    <#,2>,r0                ; check numeric args
SUBOB1: gettyp  <@,2>,r1               ;get sign
        tstb    <#,F$SIGN>,r1,SUBAOR    ;must be positive
        subvec  <@,2>,<#,NUMLEN>       ;get length
        jmpei   NUMLEN,<#,1>,SUBAOR     ; zero args are out of range
        jmpgi   NUMLEN,<#,2>,SUBAOR     ;too big
        subvec  <@,2>,<#,TMPINT>,<#,2> ;get integer
        deci    r0
        jmpzi   r0,SUBOB2               ; check another numeric arg?
        decp    r2                      ; yes, save first one in r3
        movi    TMPINT,r3
        jmp     SUBOB1
SUBOB2: jmpli   TMPINT,r3,SUBAOR        ; check for 1st arg > 2nd
        decp    r2                      ; point to third arg
        jmpep   <#,EMPTY>,<@,2>,AORER1 ; EMPTY can't be an argument
        movi    <#,1>,r1
        call    CVTSTR,GIVEUP           ;make 3rd arg. string
        gettyp  <@,2>,r1
        tstb    <#,F$LIST>,r1,SBJLST    ; check type of 3rd arg
;
; use "exts" to create a new string containing the desired
; substring of the third argument.

        getlen  <@,2>,r1                ; check length of word
        jmpgi   r3,r1,AORER1     ; error if 1st arg out of range
        jmpgi   TMPINT,r1,AORER2        ; error if 2nd arg out of range
        subi    r3,TMPINT
        inci    TMPINT                  ; TMPINT now has len. of substr.
        movp    <#,NULL>,r0             ; flush any garbage out of r0
        exts    <@,2>,r0,r3,TMPINT

SBJDON: pop     r3
        succeed

;
;  get Mth to Nth elements of a list structure
SBJLST: subi    r3,TMPINT               ; get # of 1st element in TMPINT
        movi    r3,r1                   ; and length-1 of subobj. in r3
        movi    TMPINT,r3
        movi    r1,TMPINT
        pushm   r2,r3
        movi    <#,B$STR>,SBIFLG
        movp    r2,ARGPTR
        movp    <#,GETARG>,SBIOBP
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI
;
;  skip over unwanted sublists

SBJSKP: deci    TMPINT
        jmpzi   TMPINT,SBJCPY
        movi    <#,0>,r3
        call    NXTLST
        tstb    <#,B$DONE>,SBIFLG,AORERA    ; error if not enpugh sublists
        jmp     SBJSKP                     ; otherwise keep skipping
;
;  copy the desired sublists

SBJCPY: swapp   <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG           ; init. output buffer
        movp    <#,SBOBUF>,r1
        call    BOUINI
        popm    r2,r3
        movi    r3,TMPINT                  ; TMPINT has number of sublists
        movi    <#,1>,r3                   ;... minus one.
SBJCPL: call    NXTLST
        jmpzi   r0,SBJCP1                  ; was sub-list a list itself?
        movi    <#,U$RBRK>,r2
        movp    <#,SBOBUF>,r1              ; yes, output right bracket
        call    BOUP
SBJCP1: jmpzi   TMPINT,SBJCPZ
        tstb    <#,B$DONE>,SBIFLG,AORERB         ; error..arg2 too large
        movi    <#,U$SPCS>,r2
        movp    <#,SBOBUF>,r1
        call    BOUP                       ; output list separator
        deci    TMPINT
        jmp     SBJCPL

SBJCPZ: pop     r3
        jmp     CATLST
;
;simple failure return for ELEMENT--ERROBJ set up by CVINUm

SBJERR: pop     r3
        fail

;
;   error if index arg out of range

AORERB: movp    ARGPTR,r2
        jmp     AORER2
AORERA: popm    r2,r3
AORER1: incp    r2
AORER2: incp    r2
SUBAOR: pop     r3
        movi    <#,E$AOR>,ERRCOD
        jmp     STDERR

;
;[
;routine: MEMBER — takes 2 arguments; 1st is either a single-character
;               word or a list element to be searched for within the 2nd
;               arg., which may be a word or a list.
;input regs: std. primitive.
;output regs: r0 — returns zero if the search fails; else returns the
;               index of the position of the first arg within the 2nd.
;attr: UNSAFE
;calls: NXTLST, buffer management, err regs.
;]
        entry   MEMBER

MEMBER: movp    r2,r1                   ; r2 points to 1st arg
        decp    r1                      ; r1 to the 2nd.
        jmpep   <#,EMPTY>,<@,1>,RZERO   ; return 0 if search list is empty
        gettyp  <@,1>,r0                ; is 2nd arg a list?
        tstb    <#,F$LIST>,r0,MEMLST
;
; if 2nd arg is a word, check that item to be searched for is
; a 1-character word.

        jmpep   <@,2>,<#,EMPTY>,SIZERR ; if EMPTY, return size error
        gettyp  <@,2>,r0                ; is it a word?
        tstb    <#,F$LIST>,r0,LSTERR     ; no, error
        getlen  <@,2>,r0                ; is its length 1?
        jmpnei  r0,<#,1>,SIZERR          ; no, error
;
; set up input buffer and argument pointers

MEMLST: push    r3
        movp    r1,ARGPTR
        movp    r2,TMPPTR                ; save arg pointers
        movi    <#,B$STR>,SBIFLG
        movp    <#,GETARG>,SBIOBP
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI
        movi    <#,1>,TMPINT    ; set up index of current sub-element
;
; main compare loop

MEMLP:  swapp   <#,EMPTY>,SBOOBP         ; init. output buffer
        movp    <#,SBOBUF>,r1
        movi    <#,B$STR>,SBOFLG
        call    BOUINI
        movp    ARGPTR,r0
        gettyp  <@,0>,r0                 ; create an obj. with the
        tstb    <#,F$LIST>,r0,MEMCPL     ; value of the next sub-element
```

490

```
        movp    <#,SBIBUF>,r1    ; which, for a word, is the next char...
        call    BINP
        tstb    <#,B$DONE>,SBIFLG,MEMZRO          ; if done, return 0
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        jmp     MEMCMP                    ; check to see if this is
                                          ; what we're looking for...

MEMCPL: tstb    <#,B$DONE>,SBIFLG,MEMZRO          ; return zero if no more
        movi    <#,2>,r3  ; if 2nd arg's a list, get next sub-element
        call    NXTLST
        movp    TMPPTR,r1
        gettyp  <@,1>,r2                  ; if 1st arg is a word,
        jmpnzi  r0,MEMCP1                 ; and sub-element's not
        tstb    <#,F$LIST>,r2,MEMNXT      ; look at next sub-element
        jmp     MEMCMP                    ; else, compare.
MEMCP1: jmpep   <#,EMPTY>,<@,1>,MEMCMP ; is sub. is a list and
        tstb    <#,F$LIST>,r2,MEMCMP      ; 1st arg is non-EMPTY or
        jmp     MEMNXT                    ; a word, look at next one.
;
;  compare current subelement and 1st argument

MEMCMP: movp    <#,SBOBUF>,r1
        call    BOUEND
        movp    TMPPTR,r0
        cmpsu   SBOOBP,<@,0>,MEMNXT,MEMNXT
        swapp   <#,NULL>,SBOOBP           ; if found, return curr. index
        movp    <#,NULL>,r0
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,TMPINT>,r0,<#,2>
        pop     r3
        succeed
;
;  loop and get next sub-element of 2nd argument.

MEMNXT: inci    TMPINT
        jmp     MEMLP
;
; return zero if search failed (no more elements).

MEMZRO: swapp   <#,NULL>,SBOOBP
        pop     r3
        jmp     RZERO

        endmod  LGOSTR
```

4.2.13 LGOSYS.LIL

```
        search LIl

        module LGOSYS --LOGO-S system primitives

        include CONSTS
        include FLAGS
        include ERRORS

        extrn   $PI, CURPRC, LEVEL, EMPTY
        extrn   ERRCOD
        extrn   STDERR, CMDDON
        extrn   GIVEUP, UNARG, CVTSTR
        extrn   CATINI, CATLST, CONCAT, SBOBUF, BOUP

;
; variables for DAYTIM and DATE
        vsect
NUMBUF: ptr     NULL
        ptr     NULL
NUMARG: ptr     NULL
INTBUF:
INT3:   int     0
INT2:   int     0
INT1:   int     0
TMPINT: int     0

        psect

;[
;routine: SETMON (#SETMON) system routine to set frame as monitor
;regs: none
;GLOBALS: LEVEL
        extrn TOPLEV
;attr: SAFE
;]
        entry   SETMON

SETMON: setb    <#,EV$MON>,<EVFLGS,4>
        jmpnzi  TOPLEV,CMDDON           ;if top level monitor has not yet
        movi    LEVEL,TOPLEV            ; been set, than this is it
        jmp     CMDDON
```

```
;[
;routine: UNWND,UNWNDW (UNWIND,UNWINDWITH —to previous stack frame)
;regs: std. normal primitive, r4 adjusted
;attr: UNSAFE
;calls:
        extrn   UNWIND
;global: LEVEL
;]
        vsect

NDIGIT: int     0                       ;number digit
RESULT: ptr     NULL                    ;holds result to be unwound


        psect

        entry   UNWND,UNWNDW

UNWNDW: omovp   <MWRDSZ,2>,RESULT       ;get 2nd arg.
        jmp     UNWND2

UNWND:  movp    <#,UNDEF>,RESULT

UNWND2: subvec  <@,2>,<#,NDIGIT>        ;get length of number
        jmpgi   NDIGIT,<#,2>,UNWERR     ;number too big
        jmpei   NDIGIT,<#,1>,UNWND3     ;number is 0
        gettyp  <@,2>,r0
        tstb    <#,F$SIGN>,r0,UNWERR    ;no negative numbers
        subvec  <@,2>,<#,NDIGIT>,<#,2> ;get level
        movi    NDIGIT,r1
        jmp     UNWND4

UNWND3: movi    TOPLEV,r1               ;zero means unwind to toplevel

UNWND4: jmpgi   r1,LEVEL,UNWERR         ;must unwind down
        call    UNWIND
        omovp   RESULT,r0               ;return 2nd arg.
        succeed

UNWERR: movi    <#,E$AOR>,ERRCOD        ;arg. out of range
        jmpep   <#,UNDEF>,RESULT,STDERR
        omovp   RESULT,<MWRDSZ,2>       ;replace arg. for UNWINDWITH
        jmp     STDERR

;[
;routine:$LEVEL (LEVEL) —returns stack frame level
;regs: std. arg.less primitive
;attr:UNSAFE
;CALLS:
        extrn CRNUM
;global: LEVEL
;]
        entry   $LEVEL

$LEVEL: movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,LEVEL>,r0,<#,2>
        succeed

;
;[
;routines: DAYTIM (TIME) and DATE (DATE) -- both take no args.
;       and return q list of three numbers. TIME returns the
;       hour, minute and second time of day while DATE returns
;       the year, month, and day.
;input regs: none
;outputregs: list of numbers in r0
;attr: UNSAFE
;CALLS:
;       UNARG,CVTSTR,CATINI,CATLST,CONCAT,buffer management
;]

        entry   DAYTIM

DAYTIM: push    r3
        host    $TIME
        jmp     DATLST

        entry   DATE

DATE:   push    r3
        host    $DATE

DATLST: movi    r0,INT1
        movi    r1,INT2
        movi    r2,INT3
        movi    <#,3>,r2
        cvtip   r2,r3
DATLS1: decp    r3
        jmpzi   <INTBUF,3>,DATLS2
        movi    <#,2>,r1
        call    CRNUM                   ; create numbers for the results
        movi    <INTBUF,3>,TMPINT
        rplvec  <#,TMPINT>,r0,<#,2>     ; of the host call...
        jmp     DATLS3
```

```
DATLS2: movi    <#,1>,r1                  ; check for zero number
        call    CRNUM
DATLS3: omovp   r0,<NUMBUF,3>
        deci    r2
        jmppi   r2,DATLS1
        movi    <#,3>,r1
        movp    <#,NUMARG>,r2             ; convert numbers to strings
        call    CVTSTR,GIVEUP
        call    CATINI
        movi    <#,3>,r3
DATLS4: call    CONCAT
        deci    r3
        jmpzi   r3,DATLS5
        movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2             ; produce list for output
        call    BOUP
        jmp     DATLS4
DATLS5: pop     r3
        movi    <#,3>,r1
        movp    <#,NUMARG>,r2
        call    UNARG                     ; deallocate fake arguments
        jmp     CATLST

;[
;routine: DELAY (WAIT) -- delays according to the number of milliseconds
;       specified in its one and only (numeric) argument
;input regs:    std prim.
;attr: SAFE
;]

        entry   DELAY

DELAY:  subvec  <@,2>,<#,TMPINT>,<#,1>
        jmpgi   TMPINT,<#,2>,TIMERR
        jmpnzi  TMPINT,DELAY0
        movi    <#,0>,r1
        jmp     DELAY1
DELAY0: gettyp  <@,2>,r0
        tstb    <#,F$SIGN>,r0,TIMERR     ; error if negative duration
        subvec  <@,2>,<#,TMPINT>,<#,2>
        movi    TMPINT,r1                ; call host routine if arg
DELAY1: host    $DELAY                   ; is in range
        jmp     CMDDON

TIMERR: movi    <#,E$AOR>,ERRCOD
        jmp     STDERR

;[
;routine: VRSNUM (VERSION)
;input regs: none
;output regs: returns the LOGO version number in r0.
;attr: UNSAFE
;]

        entry   VRSNUM

VRSNUM: movi    <#,VERNUM>,r0
        jmpzi   r0,VRSNU0
        movi    r0,TMPINT
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,TMPINT>,r0,<#,2>
        succeed
VRSNU0: movi    <#,1>,r1
        call    CRNUM
        succeed


        endmod  LGOSYS
```

4.2.14 PARSER.LIL

```
       search  LIL
        module  PARSE
        entry   PARSE, PEXPR

        include ERRORS
        include CONSTS
        include TOKENS
        include FLAGS
;
; external routines

        extrn   BOUINI, BOUP, BOUEND      ;buffer management
        extrn   RDOBJ, RDTAIL             ;reader
        extrn   GIVEUP
;
; external names

        extrn   PT, NT, RWT               ;name tables
        extrn   EMPTY
        Const   NUMONE,NUMFAK+1         ;phony number of args for
                                        ; unary operators
        Const   NUMTWO,NUMFAK+2         ;phony number of args for
                                        ; binary operators
        vsect

;
; PSOBUF -- output BCB for parse vector

PSOSIZ: int     PSBFSZ
PSOCNT: int     0
PSOPTR: ptr     PSOBUF
PSOOBP: ptr     null
PSOOBC: int     0
PSOOBL: int     0
PSOFLG: int     0                       ;this is a vector buffer
PSOBUF: blkp    PSBFSZ

;
; SCOBUF -- output BCB for scanner. (portion of line read is placed here)

SCOSIZ: int     SCBSIZ
SCOCNT: int     0
SCOPTR: ptr     SCOBUF
SCOOBP: ptr     null
SCOOBC: int     0
SCOOBL: int     0
SCOFLG: int     0                       ;this is a vector buffer
SCOBUF: blki    SCBSIZ
;
; parser externals

        entry   LEXRCH          ; addr. of routine to supply character

        extrn   ERRCOD,ERROBJ,ERRINF ;globals for passing error info

        extrn   CVTCN                   ;char. to number converter
;
; symbol table entries used by parser

        entry   MAKENT,DIFENT,PRDENT,REMENT,QUOENT,MINENT,LESENT
        entry   GRTENT,EQUENT,SUMENT
        entry   INSENT
;
; EXPRession proc. table entries, indexed by T$OPCD bits of operator
; tokens
```

```
PROCTB:
MAKENT: ptr     NULL
DIFENT: ptr     NULL
PRDENT: ptr     NULL
REMENT: ptr     NULL
QUOENT: ptr     NULL
LESENT: ptr     NULL
GRTENT: ptr     NULL
EQUENT: ptr     NULL
SUMENT: ptr     NULL
MINENT: ptr     NULL
;
; INSERT#LINE entry

INSENT: ptr     NULL

;
; parser & scanner variables

LEXRCH: ptr     NULL
csave:  int     0                  ; Saved character.
cflag:  int     0                  ; True if csave contains something.
qtflag: int     0                  ; Positive if last char was an @
CODPTR: ptr     0                  ; Offset in code buffer.
SCNMOD: int     0                  ; if non-zero, scanner does not look
                                   ; up unquoted words in the reserved word
                                   ; table
token:  int     0                  ; Return value from scanner.
tokval: ptr     NULL               ; Ptr to datum returned by scanner.
ENDWRD: int     0                  ; signals to SCOCHR if token has been read
NUMTOK: int     0                  ; non-zero if an integer has been read
TMPPTR: ptr     0                  ; Temporary

        psect
;
; There are two entry points for the Parser: PARSE & PEXPR.  PARSE parses
; an LSTMT, i.e., a full LOGO command line with multiple & nested statements.
; PEXPR, which is only called by the LOGO primitive VALUE, parses a single
; expression which returns a value.
;
; The calling routine (should be EVAL) passes the address of a routine to
; give characters in LEXRCH.  This routine should fail (repeatedly) on an EOL.
; The parse vector is returned in R0.
;
; the detection of a truly fatal error causes the token T$ferr to be placed
; in TOKEN, which in turn causes the scanner to refuse to work.
```

```
PARSE:  push    r1
        movi    <#,0>,NUMTOK
        movp    <#,LOGOST>,r1
        jmp     PARSER

PEXPR:  push    r1
        movp    <#,EXPR>,r1
        jmp     PARSER

PARSER: call    stemit
        call    strtch
        swapp   <#,EMPTY>,TOKVAL
        movi    <#,0>,r0
        movi    r0,TOKEN
        movi    r0,SCNMOD
        movi    r0,ENDWRD
        call    SCAN,PFAIL
        jmpei   TOKEN,<#,T$EOL>,PRSDON
        call    <@,1>, PFAIL
        jmpnei  token, <#,T$EOL>, EOLFAI
PRSDON: movp    <#,O$EOL>, r0
        call    emit
        call    NDEMIT                   ;return parse vector in r0
        swapp   <#,NULL>,SCOOBP          ;deallocate scan buffer
        pop     r1
        succeed

EOLFAI: jmpei   TOKEN,<#,T$FERR>,PFAIL   ;don't set ERRCOD if already set
        movi    <#,E$JUNK>, ERRCOD
PFAIL:  movp    <#,PSOBUF>,r1            ;move all heap pointers out
        call    BOUEND                   ; of buffer
        jmpep   <#,NULL>,TOKVAL,PFAIL1

PFAIL1: swapp   <#,NULL>,PSOOBP          ;deallocate any partially created
        movp    <#,SCOBUF>,r1
        call    BOUEND                   ;close up scan buffer
        swapp   SCOOBP,ERROBJ            ; for error context
        gettyp  ERROBJ,r1
        setb    <#,F$LIST>,r1
        settyp  r1,ERROBJ
        pop     r1                       ; parse vector
        fail
;
;LOGOST: Lstatement
;       Number   ANY-STRING-OF-CHARACTERS

LOGOST: jmpnzi  NUMTOK,IN$LIN
;
```

```
;fall thru into Lstatement if line doesn't begin with number
;
; LStatement:   IfSeq
;               IterSeq
;               ";" [ Lstatement ]
;               Comand [ ";" [ Lstatement ] ]
;

LSTMT:  jmpei   TOKEN,<#,T$IF>,IFSEQ     ;return thru IFSEQ
        call    ITRSEQ,LSTMT1
        succeed

LSTMT1: call    COMAND,LSTMT2
        jmpei   TOKEN,<#,T$SEMI>,LSTMT3
        succeed

LSTMT2: jmpei   TOKEN,<#,T$SEMI>,LSTMT3
        movi    <#,E$EXEX>,r0    ;expression expected
        jmp     FATERR
;
;semi-colon recognized

LSTMT3: call    SCAN,PSERR
LSTMT4: jmpei   <#,T$SEMI>,TOKEN,LSTMT3
        jmpei   TOKEN,<#,T$IF>,IFSEQ
        call    ITRSEQ,LSTMT5
        succeed

LSTMT5: call    COMAND,LSTDON
        jmpei   TOKEN,<#,T$SEMI>,LSTMT3
LSTDON: succeed

;
;INSLIN —read in rest of line & pass line number, text as args. to
;       INSERT#LINE

INSLIN: pushm   r2,r3
        movp    <#,O$LITC>,r0
        call    EMIT
        movp    TOKVAL,r0
        call    EMIT                     ;emit line number literal
;
;read in line through SCOBUF

        swapp   <#,NULL>,SCOOBP          ;deallocate any previous junk in SCOBUF
        call    STRTCH                   ;re-init SCOBUF
        movp    LEXRCH,r3                ;get character routine
INSLP:  call    <@,3>,INSDON            ;read until EOL
```

```
        movi    r0,r2
        movp    <#,SCOBUF>,r1
        call    BOUP                    ;output char.
        jmp     INSLP

INSDON: movp    <#,SCOBUF>,r1
        call    BOUEND
        movp    <#,O$LITC>,r0
        call    EMIT
        movp    SCOOBP,r0
        call    EMIT                    ;output code for rest of line
;
; output code for call to INSERT#LINE

        movp    <#,O$NCAL>,r0
        call    EMIT
        movp    INSENT,r0
        call    EMIT
        movp    <#,NUMTWO>,r0
        call    EMIT
        movp    <#,O$EOS>,r0
        call    EMIT
        popm    r2,r3
        movi    <#,T$EOL>,TOKEN
        succeed

;
; IterSeq:  WhileStmt
;           UntilStmt
;           RepeatStmt
;           IterateStmt

ITRSEQ: jmpei   token, <#,T$WHIL>, whstmt
        jmpei   token, <#,T$UNTL>, unstmt
        jmpei   token, <#,T$REPT>, rpstmt
        jmpei   token, <#,T$ITER>, itstmt
        fail
;
; Ifseq: IF expr Do IfBody [ELSE LStatement]

IFSEQ:  push    r3
        call    SCAN,IFFAIL
        call    expr, IFFAIL
        movp    <#,O$JMPI>, r0
        call    emit
        call    mark
        call    dohick, IFFAIL
        call    IFBODY, IFFAIL
        jmpnei  token, <#,T$ELSE>, isl
        push    r1
        movp    <#,O$JUMP>, r0
        call    emit
        movi    r3,r1                   ;save IF part patch address
        call    mark
        call    SCAN,IFFAI2
        push    r3                      ;save ELSE part patch addr.
        movi    r1,r3                   ;& restore IF addr.
        call    patch
        pop     r3                      ;restore ELSE addr.
        pop     r1
        call    lstmt, IFFAIL
isl:    call    patch
        pop     r3
        succeed

IFFAI2: pop     r1
IFFAIL: pop     r3
        fail
;IterTail
;IfBody         IterSeq
;               ";" [ IfBody ]
;               Command [ ";" [ IfBody ]  ]
;(IterTail & IfBody are the same, in V.0)

ITRTAI:
IFBODY: call    ITRSEQ,IFBOD1
        succeed

IFBOD1: call    COMAND,IFBOD2
        jmpei   <#,T$SEMI>,TOKEN,IFBOD3
        succeed

IFBOD2: jmpnei  TOKEN,<#,T$SEMI>,PSERR
;
;semi-colon recognized

IFBOD3: call    SCAN,PSERR
IFBOD4: jmpei   <#,T$SEMI>,TOKEN,IFBOD3
        call    ITRSEQ,IFBOD5
        succeed

IFBOD5: call    COMAND,IFBDON
        jmpei   TOKEN,<#,T$SEMI>,IFBOD3
IFBDON: succeed

;
```

```
; WhileStmt: WHILE Loopbody

whstmt: push    r1
        movi    <#,O$JMPW>,r1
        call    LPBODY,CNDFAI
        pop     r1
        succeed

; UntilStmt: UNTIL Loopbody

UNSTMT: push    r1
        movi    <#,O$JMPU>,r1
        call    LPBODY,CNDFAI
        pop     r1
        succeed

; RepeatStmt: REPEAT Loopbody

RPSTMT: push    r1
        movi    <#,O$DECJ>,r1
        call    LPBODY,CNDFAI
        pop     r1
        succeed
;
;CNDFAI —common fail for conditional loops

CNDFAI: pop     r1
        fail
;
;[
;routine: LPBODY —  expr Do IterTail
;input reg: r1 —OM code for type of loop
;attr: UNSAFE
;]

LPBODY: push    r3
        jmpep   r1,<#,O$DECJ>,LPBOD1
        movi    CODPTR,r3                 ;save addr. of cond. expr.
                                          ;for WHILE & UNTIL loops
LPBOD1: call    SCAN,LPFAI2               ;(expr must be evaluated each
        call    expr, LPFAI2              ; time)
        call    dohick, LPFAI2
        jmpnep  r1,<#,O$DECJ>,LPBOD2      ;save addr. of loop for
        movi    CODPTR,r3                 ;REPEAT (expr only evaluated
                                          ;once)
LPBOD2: pushi   r3                        ;push CODPTR value
        movp    r1, r0
        call    emit
        call    mark
        call    ITRTAI, LPFAIL
        movp    <#,O$JUMP>,r0
        call    emit
        popi    r0                      ;get old CODPTR
        addp    <#,NUMFAK>,r0           ;make phony ptr.
        call    EMIT
        call    patch                   ;now adjust conditional jump
        pop     r3
        succeed

LPFAIL: popi    r0                      ;throw away CODPTR
LPFAI2: pop     r3
        fail

; IterateStmt: ItHead Do StmtSeq
; ItHead: ITERATE Expr FROM expr ItCond Expr
; ItCond: WHILE | UNTIL

itstmt: ; fill in later...much later
        fail

; Do: DO | THEN | ","

dohick: jmpei   token, <#,T$DO>, dol
        jmpei   token, <#,T$COMM>, dol
        movi    <#,E$DOEX>, r0
        jmp     FATERR                  ;fatal error

dol:    call    SCAN,PSERR
        succeed

;Command        "{" Lstmt "}"
;               Expr
;               "{" "}"

 COMAND:        jmpei   token, <#,T$LBRC>, sil
        call    expr, pserr
        movp    <#,O$EOS>, r0
        call    emit
        succeed

sil:    call    SCAN,PSERR
        jmpei   token, <#,T$RBRC>, COMDON ;if {}, then generate nothing
        call    lstmt, SI1ERR
        jmpei   token, <#,T$RBRC>,COMDON
SI1ERR: movi    <#,E$RBRC>,r0
        jmp     FATERR                  ;fatal error
```

```
COMDON: call    SCAN,PSERR
COMEND: succeed


;
;Expression Parsing
;
;[
;ROUTINE:BINOPR--determines if the current token is a binary operator
;               of the desired type (relational ,additive, etc.)
;input REGS: r2 -- type desired
;output REGS:r1 -- current token, clobbers r0
;               fails if token is not correct
;ATTR:SAFE
;TYPE:local utility
;]

BINOPR: movi    TOKEN,r1
        tstb    <$,T$BOPR>,r1,BINOP2
        jmp     BINFAI                  ;fail if token not operator
BINOP2: movi    <$,T$TYPE>,r0           ;mask for type bits of token code
        andi    r1,r0
        jmpnei  r0,r2,BINFAI            ;wrong type
        call    SCAN,BINFAI             ;grab next token
        succeed

BINFAI: fail

;EXPR   EXP1 [ <- EXP1 ]
;

EXPR:   tstb    <$,T$EXPR>,TOKEN,EXPR1  ;check if any chance of finding EXPR
        fail                            ; if not, fail right away

EXPR1:  pushm   r1,r2
        call    EXP1,EXPFAT
        movi    <$,T$ASGN>,r2
        call    BINOPR,EXPRON           ;check for assignment operator
        call    EXP1,EXPFAT
        call    EXEMIT                  ;output call to MAKE
EXPRON: popm    r1,r2
        succeed
;
;fatal error in EXPRession

EXPFAT: popm    r1,r2
        movi    <$,E$EXEX>,r0
        jmp     FATERR
;
;EXP1   EXP2 [ RELOP EXP2 ]
;
EXP1:   push    r1
        call    EXP2,EXPFAI
        movi    <#,T$REL>,r2
        call    BINOPR,EXPDON           ;check for relational operator
        call    EXP2,EXPFAI
        call    EXEMIT
        jmp     EXPDON
;
;EXP2   EXP3 [ ADDOP EXP3 ]*

EXP2;   push    r1
        call    EXP3,EXPFAI
EXP2A:  movi    <#,T$ADD>,r2
        call    BINOPR,EXPDON
        call    EXP3,EXPFAI
        call    EXEMIT
        jmp     EXP2A
;
;EXP3   PRIMARY [ MULOP PRIMARY ]*
;
EXP3:   push    r1
        call    PRIMRY,EXPFAI
EXP3A:  movi    <#,T$MUL>,r2
        call    BINOPR,EXPDON
        call    PRIMRY,EXPFAI
        call    EXEMIT
        jmp     EXP3A

;
;all EXPn succeed thru here

EXPDON: pop     r1
        succeed
;
; EXPn failure

EXPFAI: pop     r1
        fail
;
; EXEMIT --emit procedure call corresponding to token in r1

EXEMIT: andi    <#,T$OPCD>,r1           ;get index nto proc. table
        cvtip   r1,r1
        movp    <#,O$NCAL>,r0
        call    EMIT
        movp    <PROCTB,1>,r0
        call    EMIT
        movp    <#,NUMTWO>,r0           ;2 args. to procedure
        call    EMIT
        return
;
; primary

PRIMRY: movi    TOKEN,r0
        jmpei   r0, <#,T$DOTS>, RFRNCE
        jmpei   R0, <#,T$LITR>, LTERAL
        jmpei   R0, <#,T$OPEN>, BLOCK1
        jmpei   R0,     <#,T$UMIN>,UMINUS
        jmp     FNCALL
;
; REFERENCE     [ : ]+ word

RFRNCE: push    r1
        movi    <#,0>, r1               ; count of # of dref's needed
REF1:   call    SCAN,REFER2
        jmpei   token, <#,T$WORD>, REFVAR
        jmpnei  token, <#,T$DOTS>, REFERR
        inci    r1
        jmp     REF1
REFVAR: movp    <#,O$VALC>, r0
        call    emit
;
; get or create name table entry for variable reference

        enter   nt, tokval, r0,PRIVAR
        setval  <#,UNDEF>,r0            ;if var. is just created, set its
                                        ;value to UNDEFined
PRIVAR: call    emit
;
;Emit calls to DREF for all dots after first

REFEMT: jmpzi   r1, REFDON
        movp    <#,O$DREF>, r0
        call    emit
        deci    r1
        jmp     REFEMT

REFDON: pop     r1
        jmp     PRMDON

REFERR: movi    <#,E$BDRF>, r0
```

```
REFER2: pop     r1
        jmp     FATERR
;
;LITERAL (number, word, list)

LTERAL: movp    <#,O$LITC>, r0
        call    emit
        movp    tokval, r0
        call    emit
        jmp     PRMDON

BLOCK1: call    SCAN,PSERR
        call    expr,BLKERR
        jmpei   token, <#,T$CLOS>, PRMDON
BLKERR: movi    <#,E$CLOS>, r0
        jmp     FATERR                  ;fatal error
;
;umary minus found--must be followed by a primary

UMINUS: call    SCAN,PSERR
        call    PRIMRY,UMNERR
        movp    <#,O$NCAL>,r0
        call    EMIT
        movp    MINENT,r0
        call    EMIT
        movp    <#,NUMONE>,r0
        call    EMIT                    ;fall into PRMDON
        succeed

UMNERR: movi    <#,E$EXEX>,r0
        jmp     FATERR

;
; general succeed & get next token for primaries

PRMDON: call    SCAN,PSERR
        succeed
;
; FNCALL
; r0- procedure table entry flags
; r1- for # of args
; r2- address of procedure table entry

FNCALL: pushm   r0,r2
        jmpnei  TOKEN, <#,T$WORD>,FNFAIL
        find    PT,TOKVAL,r2            ;get addr. of PT entry
        jmpep   <#,NULL>,r2,FNERR1      ;jump if proc. non-existent
        getval  r2,r0
        jmpep   r0,<#,UNDEF>,FNERR1     ;proc. has been erased

        getflg  r2,r0
        tstb    <#,F$GRDY>,r0,FNGRDY
        movi    r0,r1
        andi    <#,F$ARGS>,r1           ;mask out all but # of args
        pushi   r1                      ;save for emitting NCAL

        tstb    <#,F$QUOT>,r0,FNQUOT

;
; get args. for normal procedures

        call    SCAN,FNFAI2
FNNORM: jmpzi   r1,FNDONE
        call    EXPR, FNFAI3
        deci    r1
        jmp     FNNORM
;
; get args. for greedies

FNGRDY: movi    <#,0>,r1                ;r1 will have count of args. scanned
        tstb    <#,F$QUOT>,r0,FNGRQU    ;jump if greedy quoting
        call    SCAN,FNFAIL
FNGRD1: call    EXPR,FNGRD2             ;grab all args. available
        inci    r1
        jmp     FNGRD1

FNGRD2: pushi   r1                      ;save arg. count
        jmp     FNDONE

;
;get args for quotings

FNQUOT: movi    <#,1>,SCNMOD    ;signal scanner to quote all words
FNQUO1: jmpzi   r1,FNQUO3
        call    SCAN,FNFAI2
        jmpei   TOKEN,<#,T$WORD>,FNQUO2
        jmpei   TOKEN,<#,T$LITR>,FNQUO2
;
; bad quoting arg.

        movi    <#,0>,SCNMOD
        popi    r0                      ;kill arg. count
        popm    r0,r2
        movi    <#,E$TFA>, r0   ;too few good args.
        jmp     FATERR
```

```
FNQUO2: movp    <#,O$LITC>,r0
        call    EMIT
        movp    TOKVAL,r0
        call    EMIT
        deci    r1
        jmp     FNQUO1

FNQUO3: movi    <#,0>,SCNMOD
        call    SCAN,FNFAI2
        jmp     FNDONE
;
;get args for greedy quotings

FNGRQU: movi    <#,1>,SCNMOD            ;signal scanner to quote all words
        movi    <#,0>,r1                ;r1 counts args. read
FNGRQ1: call    SCAN,FNFAIL
        jmpei   TOKEN,<#,T$WORD>,FNGRQ2
        jmpei   TOKEN,<#,T$LITR>,FNGRQ2
;
;non-quotable token read—end arg. scan

        pushi   r1                      ;push # of args.
        movi    <#,0>,SCNMOD
        jmp     FNDONE

FNGRQ2: movp    <#,O$LITC>,r0
        call    EMIT
        movp    TOKVAL,r0
        call    EMIT
        inci    r1
        jmp     FNGRQ1
;
; emit CM code to call procedure

FNDONE: movp    <#,O$NCAL>,r0
        call    EMIT
        movp    r2,r0                   ;procedure table entry
        call    EMIT
        popi    r0                      ;pop no. of args.
        addp    <#,NUMFAK>,r0           ;make phony ptr.
        call    EMIT
        popm    r0,r2
        succeed

FNERR1: popm    r0,r2
        movi    <#,E$FNEX>, r0
        jmp     FATERR
```

```
FNFAI3: popi    r0                      ;kill arg. count push
        popm    r0,r2
        movi    <#,E$TFA>,r0            ;too few args.
        jmp     FATERR

FNFAI2: popi    r0                      ; kill arg. count push for

FNFAIL: popm    r0,r2
        fail
;
;[
;ROUTINE: FATERR—general failure exit for parsing routines
;         PSERR —entry point for simple fails
;input REG: r0—error code
;]
FATERR: jmpei   TOKEN,<#,T$FERR>,PSERR  ;don't set ERRCOD if already set
        movi    r0,ERRCOD
        movi    <#,T$FERR>,TOKEN        ;signal fatal error, stop scanning
PSERR:  fail
;
; Code emission routines

;[
;routine:  STEMIT—initialises the code emission package.
;attr: UNSAFE
;globals: PSOBUF BCB, CODPTR
;calls: BOUINI, BOUP
;]

STEMIT: pushm   r1,r2
        swapp   <#,NULL>,PSOCBP
        movi    <#,0>,PSOFLG            ;mark PSOBUF as vector buffer
        movp    <#,PSOBUF>,r1
        call    BOUINI
        movi    <#,1>,r2                ;set up NINTS of parse vector
        call    BOUP                    ;(always 1)
        movi    <#,2>,CODPTR            ;offset within parse vector
        popm    r1,r2
        return                          ;of 1st CM code

;[
;routine: MARK — marks a forward jump address (offset)
;output reg: r3 — gets current offset, r0 destroyed
;global: CODPTR
;attr: SAFE
;]

MARK:   movi    CODPTR, r3
```

```
        movp    <#,O$NOOP>, r0
         call   emit
        return
;[
;routine: PATCH — replace previously MARKed jump offset with CODPTR
;               if offset is still in buffer, just adjust buffer contents
;               else replace offset in parse vector
;input reg:r3 —marked address (offset)
;global: CODPTR
;attr: SAFE
;]

PATCH:  movi    CODPTR,TMPPTR           ;make current offset
        addp    <#,NUMFAK>,TMPPTR       ;into phony ptr.
;
; calc. if marked ptr. is in PSOBUF

        movi    r3,r0
        subi    PSOOBC,r0               ;get offset within buffer of
        jmpmi   r0,PATCH1               ;marked ptr.(if negative,
                                        ;then written out already
;
;adjust buffer

        cvtip   r0,r0
        movp    TMPPTR,<PSOBUF,0>
        return
;
;adjust parse vector

PATCH1: rplvec  <#,TMPPTR>,PSOOBP,r3
        return
;[
;routine: EMIT — outputs OM code
;input reg: r0 —pointer to be emitted
;globals: CODPTR,PSOBUF BCB
;calls: BOUP
;attr: UNSAFE
;]

EMIT:   pushm   r1,r2
        movp    r0,r2
        movp    <#,PSOBUF>,r1
        call    BOUP
        inci    CODPTR
        popm    r1,r2
        return
;[
;routine: NDEMIT --closes up PSOBUF
;output reg: r0 --parse vector
;attr: UNSAFE
;calls: BOUEND
;globals: PSOOBP
;]

NDEMIT: push    rl
        movp    <#,PSOBUF>,rl
        call    BOUEND
        omovp   PSOOBP,r0
        pop     rl
        return

;
;[
;routine: SCAN
;]

SCAN:   jmpei   TOKEN,<#,T$ferr>,PSERR  ;don't work if fatal error detected
        push    rl
sc01:   call    nextch
        jmpei   rl, <#,T$BLNK>, sc01
        jmpei   rl,<#,T$CHRQ>,SCNWRD    ; if first char is a super-
                                        ; quote ("@"), read in a word.
        jmpei   rl,<#,T$CMNT>,SCNCMT
        jmpei   rl, <#,T$IDCH>, SCNWRD
        jmpei   rl,<#,T$LBRK>,SCNLST
        jmpnzi  SCNMOD,SCNQWD           ;always get word or list in
                                        ;quote argument mode
        jmpei   rl,<#,T$UMIN>,SCNMIN
        jmpei   rl,<#,T$DIGIT>,SCNNUM
        jmpei   rl,<#,T$QUOT>,SCNQUO
SCNTOK: movi    rl, token
        jmp     SCNDON

SCNQWD: tstb    <#,T$PUNC>,rl,SCNTOK    ;in quoting mode, check if
                                        ;char. can start a quoting
                                        ;if not, return token

SCNWRD: call    PUTBAK                  ;make sure 1st char is read
        call    GETWRD                  ; by GETWRD
        find    RWT,r0,rl
        swapp   r0,TOKVAL
        jmpnzi  SCNMOD,SC04             ;don't look for resvd words in
                                        ;quoting mode
        jmpep   rl,<#,NULL>,SC04        ;not a reserved word
        getflg  rl,TOKEN
```

513

```
        jmp     SCNDON

SC04:   movi    <#,T$WORD>, TOKEN
        jmp     SCNDON
;
; read the tail of a list (left bracket already read in)

SCNLST: movp    <#,LSTCHR>,r1
        call    RDTAIL
        jmpep   r0,<#,UNDEF>,SCLERR     ;ill-formed list read
        swapp   r0,TOKVAL
        movi    <#,T$LITR>,TOKEN
        jmp     SCNDON
;
; quote found, read in word

SCNQUO: call    GETWRD
        swapp   r0,TOKVAL
        movi    <#,T$LITR>,TOKEN
        jmp     SCNDON
;
; digit found, read in number

SCNNUM: movi    <#,1>,NUMTOK            ;signal number read for line insertions
        call    PUTBAK
        movp    <#,WRDCHR>,r1           ;routine to read in digits
        call    CVTCN,SCNNER            ;read number or fail
        swapp   r0,TOKVAL
        movi    <#,T$LITR>,TOKEN
        jmp     SCNDON

SCNNER: movi    <#,E$BFN>,ERRCOD
        jmp     SCNERR
;
;minus sign found, check if unary or binary

SCNMIN: call    NXTQCH
        jmpei   r1,<#,T$BLNK>,SCNDFF    ;blank follows, so binary
        call    PUTBAK
        movi    <#,T$UMIN>,TOKEN
        jmp     SCNDON

SCNDFF: movi    <#,T$DIFF>,TOKEN
        jmp     SCNDON
;
; ill-formed list errors, error code in r1

SCLERR: movi    r1,ERRCOD       ;otherwise, extra right bracket
```

```
;
;scanner errors come here

SCNERR: pop     r1
        movi    <#,T$ferr>,TOKEN
        fail

SCNDON: movi    <#,1>,ENDWRD             ;mark end of token for SCOCHR
        pop     r1
        succeed
;
; comment found—read until end of comment or EOL

SCNCMT: call    NXTQCH
        jmpei   r1,<#,T$CMNT>,SC01
        jmpnei  r1,<#,T$EOL>,SCNCMT
        movi    r1,TOKEN                 ;return EOL token if found
        jmp     SCNDON

;
; Character routines
;[
;routine: WRDCHR —supplies chars. to reader for reading words
;               fails on puncutation char.
;output reg: r0
;attr: SAFE
; globals: NEXTCH, PUTBAK
;]

WRDCHR: push    r1
        call    NEXTCH
        jmpei   r1,<#,T$EOL>,WRDEOL
        jmppi   QTFLAG,WRDQCH            ; handle quoted character
        jmpnei  r1,<#,T$CHRQ>,WRDTRM     ; if this is a character quote,
        movi    <#,1>,QTFLAG             ; set flag for next character.
        jmp     WRDRET                   ; and return the char quote itself.
WRDTRM: tstb    <#,T$PUNC>,r1,WRDEND
        jmp     WRDRET

WRDQCH: movi    <#,0>,QTFLAG             ; reset flag after quoted char.

WRDRET: pop     r1
        succeed

WRDEND: call    PUTBAK

WRDEOL: pop     r1
        fail
```

```
;[
;routine: LSTCHR —supplies chars. to reader for reading lists
;               fails on EOL
;output reg: r0
;attr: UNSAFE
;calls: BOUP
;global: LEXRCH
;]

LSTCHR: push    r1
        movp    LEXRCH,r1
        call    <@,1>,LSTEOL
;
; output newly read char.

        call    SCOCHR
        pop     r1
        succeed

LSTEOL: pop     r1
        fail
;
;[
;routine: GETWRD —gets word from reader(terminated by puncuation)
;output reg:r0
;attr: UNSAFE
;calls: RDOBJ, WRDCHR
;called by: SCAN
;]

GETWRD: push    r1
        movi    <$,0>,QTFLAG            ; initialize character-quote flg
        movp    <#,WRDCHR>,r1
        call    RDOBJ
        jmpep   <#,UNDEF>,r0,GIVEUP
        pop     r1
        return

;[
;routine NXTQCH — calls NEXTCH; if a character-quote is returned,
;                 returns the result if a second call to NEXTCH;
;                 otherwise returns the character read initially.
;]

NXTQCH: call    NEXTCH
        jmpnei  r1,<#,T$CHRQ>,NXQDON
        call    NEXTCH
NXQDON: return
```

```
;[
;ROUTINE:SCOCHR--builds list of tokens scanned in SCOBUF
;input REG:r0--char. to be output to SCOBUF
;ATTR:UNSAFE
;CALLS:BOUP
;GLOBAL: ENDWRD
;]

SCOCHR: pushm   r0,r2
        movp    <#,SCOBUF>,r1
        jmpzi   ENDWRD,SCOCH2           ;if last char. output was end of token
        movi    <#,U$SPCS>,r2           ; output list separator
        pushi   r0
        call    BOUP
        movi    <#,0>,ENDWRD
        popi    r0

SCOCH2: movi    r0,r2
        call    BOUP
        popm    r0,r2
        return

;[
;  NEXTCH
;
;   Read a character via ´lexrch´, and return it in
;   r0. Put its type into r1.
;   All chars. read for the first time are output to SCOBUF
;]

NEXTCH: jmpnzi  cflag, nx01
        movp    lexrch, r1
        call    <@,1>,NXTEOL   ;LEXRCH fails on EOL

        call    chtype
        jmpei   r1,<#,T$BLNK>,NXTDON
;
; output newly read char. if not a white space

        call    SCOCHR

NXTDON: return
;
;return saved char.

nx01:   movi    csave, r0
        movi    <#,0>, cflag
        call    chtype
        return

NXTEOL: movi    <#,T$EOL>,r1
        movi    <#,0>,r0
        return

PUTBAK: movi    <#,1>, cflag
        movi    r0, csave
        return
;[
;routine:STRTCH —initialize scan buffer,NEXTCH flag (CFLAG)
;calls: buffer mangagement
;attr: UNSAFE
;]

STRTCH: push    r1
        movi    <#,0>, cflag
        movi    <#,B$STR>,SCOFLG
        swapp   <#,EMPTY>,SCOOBP
        movp    <#,SCOBUF>,r1
        call    BOUINI
        pop     r1
        return
;[
;  CHTYPE
;
;  This routine puts a code for the character
;   found in r0 into r1. It is separated from the rest
;   of the scanner so that it may be optimised for particular
;   character codes.
; (this routine is called from outside the parser by TRNOBJ in the
;   I/O package.)
;]
        entry   CHTYPE

CHTYPE: jmpnei  r0, <#,SPACE>, ct01
        movi    <#,T$BLNK>, r1
        jmp     ct99
ct01:   jmpnei  r0, <#,TAB>, ct02
        movi    <#,T$BLNK>, r1
        jmp     ct99
ct02:   jmpnei  r0, <#,LF>, ct05
        movi    <#,T$BLNK>, r1
        jmp     ct99
ct05:   jmpnei  r0, <#,C$SEMI>, ct06
        movi    <#,T$SEMI>, r1
        jmp     ct99
```

```
ct06:   jmpnei  r0, <#,C$DOTS>, ct07
        jmpnzi  SCNMOD,CT07                 ;do nothing special for dots in
        movi    <#,T$DOTS>, r1              ; quoting mode
        jmp     ct99
ct07:   jmpnei  r0, <#,C$QUOT>, ct08
        movi    <#,T$QUOT>, r1
        jmp     ct99
ct08:   jmpnei  r0, <#,C$OPEN>, ct09
        movi    <#,T$OPEN>, r1
        jmp     ct99
ct09:   jmpnei  r0, <#,C$CLOS>, ct10
        movi    <#,T$CLOS>, r1
        jmp     ct99
ct10:   jmpnei  r0, <#,C$LBRK>, ct11
        movi    <#,T$LBRK>, r1
        jmp     ct99
ct11:   jmpnei  r0, <#,C$RBRK>, ct12
        movi    <#,T$RBRK>, r1
        jmp     ct99
ct12:   jmpnei  r0, <#,C$LBRC>, ct13
        movi    <#,T$LBRC>, r1
        jmp     ct99
ct13:   jmpnei  r0, <#,C$RBRC>, ct14
        movi    <#,T$RBRC>, r1
        jmp     ct99
ct14:   jmpnei  r0, <#,C$COMM>, ct15
        movi    <#,T$COMM>, r1
        jmp     CT99
ct15:   jmpnei  r0,<#,C$CHRQ>, CTCMNT
        movi    <#,T$CHRQ>,r1
        jmp     CT99
CTCMNT: jmpnei  r0,<#,C$CMNT>,CTUMIN
        movi    <#,T$CMNT>,r1
        jmp     CT99

CTUMIN: jmpnei  r0,<#,C$MINS>, CTNUMB
        movi    <#,T$UMIN>,r1
        jmp     ct99
;
; scan a digit

CTNUMB: movi    r0,r1
        cvtdi   r1,CTIDCH                   ;jump if no digit
        movi    <#,T$DIGIT>,r1
        jmp     CT99

CTIDCH: movi    <#,t$IDCH>, r1
ct99:   return
```

519

```
        endmod  parse
```

4.2.15 READER.LIL

```
        search  LIL

        module  READER  ;all routines which create LOGO objects from
                        ; character streams
        include FLAGS
        include CONSTS
        include ERRORS

; reader output buffer

        vsect

RDOSIZ: int     RBUFSZ
RDOCNT: int     0
RDOPTR: ptr     RDOBUF
RDOOBJ: ptr     null
RDOOBC: int     0
RDOOBL: int     0
RDOFLG: int     B$STR
RDOBUF: blki    RBUFSZ

; reader input buffer

RDISIZ: int     RBUFSZ
RDICNT: int     0
RDIPTR: ptr     RDIBUF
RDIOBJ: ptr     null
RDIOBC: int     0
RDIOBL: int     0
RDIFLG: int     B$STR
RDIBUF: blki    RBUFSZ

        extrn   BOUINI, BOUP, BOUEND    ;buffer routines
        extrn   BININI,BINP
        extrn   EMPTY,ERRCOD,ERROBJ
        extrn   STDERR,REMPTY
        psect
;
```

```
;[
;routine: RDINIT —initializes reader
;type: called by RDOBJ, RDTAIL
;globals: LEVEL , EMPTY, LASTRB
;input reg: r1— contains addr. of read char. routine
;output reg: r3 — get addr. of routine
;attr: SAFE
;]

RDINIT: movp    <#,EMPTY>,RDOOBJ          ; output object initially is
                                          ;  EMPTY
        movi    <#,B$STR>,RDOFLG
        movp    r1,r3
        movp    <#,RDOBUF>,r1
        call    BOUINI
        movi    <#,0>,LEVEL
        movi    <#,0>,LASTRB
        return

;[
;routine: GETSIG —looks for next significant char. in reader input stream
;               fails if end of stream reached
;input reg: r3 — addr. of get char. routine
;output reg: r0 — sign. char.
;attr: SAFE
;called by RDOBJ
;]

GETSIG: call    <@,3>,GETEND
        jmpei   <#,SPACE>,r0,GETSIG
        succeed

GETEND: fail

;
;[
;routine: RDOBJ — reads first LOGO object encountered, word or list
;         RDTAIL is alternate entry point when a left bracket has
;               already been encountered
;input reg: r1— contains addr. of read char. routine
;output reg: r0 — the object read, in internal form
;              returns UNDEF if unmatching left & right brackets
;           r1 — the last char. read in if a word is returned
;                 0 if a list is returned
;              if error, returns error code
;type: utility
;calls: buffer management, RDINIT
;globals: RDOBUF bcb
;attr: UNSAFE
;]
        entry   RDOBJ,RDTAIL

        vsect

LEVEL:  int     0                       ; depth of list
LASTRB: int     0                       ; non-zero if last char. read
        psect                           ; was right bracket

RDOBJ:  pushm   r2,r3
        call    RDINIT                  ; initialize RDOBUF, LEVEL, r3
;
;search for the next significant char. of a word or list

RD1ST:  call    GETSIG,RDEND            ; get next non-blank
        jmpei   r0,<#,"]">,RDLEND       ; end of list found

        jmpzi   LASTRB,RD1ST1
        pushi   r0
        movi    <#,U$SPCS>,r2           ; if last char. read was right brk,
        movp    <#,RDOBUF>,r1           ;  output separator
        call    BOUP
        movi    <#,0>,LASTRB
        popi    r0

RD1ST1: jmpei   r0,<#,"[">,RDTAI1       ; go read tail of list
                                        ; else get a word
        jmpei   r0,<#,"@">,RDWRDQ      ; if initial char of word is
                                        ; quoted, don't store quote chr.
;
; build word

RDWRD:  movi    r0,r2                   ; add char. to word
        movp    <#,RDOBUF>,r1
        call    BOUP
;
;get next char.

        call    <@,3>,RDEND            ; jump if no more chars.
        jmpnei  r0,<#,"@">,RDWRD0      ; if  char is a quote character,
RDWRDQ: call    <@,3>,RDEND            ; then get next character
        jmp     RDWRD                   ; but don't check for [ ] or spc
RDWRD0: jmpei   r0,<#,SPACE>,RDWRDS
        jmpei   r0,<#,"[">,RDWRDL
        jmpei   r0,<#,"]">,RDWRDR
        jmp     RDWRD
```

```
;
; word terminated with a space

RDWRDS: jmpzi   LEVEL,READWD            ; if word is not in sub-list,
                                        ;  then return
        call    GETSIG,RDEND            ; else find next non-blank
        jmpnei  r0,<#,"@">,RDWRD1      ; check for quoting character
        call    <@,3>,RDEND            ; if present,
        jmp     RDWRD2                  ; ...begin a new word
RDWRD1: jmpei   r0,<#,"]">,RDWRDR       ; this ends sub-list
        jmpei   r0,<#,"[">,RDWRDL       ; this starts new sub-list
;
; word followed by another word

RDWRD2: pushi   r0                      ; save 1st char. of new word
        movi    <#,U$SPCS>,r2
        movp    <#,RDOBUF>,r1
        call    BOUP                    ; otherwise, output sub-list separator
        popi    r0                      ; restore char.
        jmp     RDWRD                   ; next item is word

; word followed by a sub-list

RDWRDL: jmpzi   LEVEL,READWD            ; return if not sub-list
        movi    <#,U$SPCS>,r2
        movp    <#,RDOBUF>,r1
        call    BOUP                    ; otherwise, output sub-list separator
        jmp     RDTAI2                  ; & read nested sub-list
;
; word terminated with right bracket

RDWRDR: jmppi   LEVEL,RDLEND            ; terminate sub-list
                                        ; else return if not sub-list
;
; return a word

READWD: pushi   r0                      ; save last char. read
        movp    <#,RDOBUF>,r1
        call    BOUEND                  ; close output buffer
        omovp   RDOOBJ,r0
        popi    r1                      ; return last char. read
        jmp     RDDONE                  ; clean up & return
;
;
; RDTAIL --entry point for lists whose left bracket has been read

RDTAIL: pushm   r2,r3
        call    RDINIT
```

```
;
; start of list or sub-list

RDTAI1: jmpzi   LEVEL,RDTAI3
RDTAI2: movi    <#,U$LBRK>,r2           ; output internal left bracket
        movp    <#,RDOBUF>,r1           ;  for sub-lists only
        call    BOUP

RDTAI3: inci    LEVEL
        jmp     RD1ST
;
; end of list or sublist

RDLEND: deci    LEVEL
        jmpmi   LEVEL,RDERR1            ; error if extra right bracket
        jmpzi   LEVEL,READLS            ;return list if brackets match
        movi    <#,U$RBRK>,r2
        movp    <#,RDOBUF>,r1
        call    BOUP                    ;else output end of sub-list char.
        movi    <#,1>,LASTRB            ; signal right bracket found
        jmp     RD1ST                   ;and keep reading
;
; return a list

READLS: movp    <#,RDOBUF>,r1
        call    BOUEND                  ; close output buffer
        omovp   RDOOBJ,r0
        jmpep   r0,<#,EMPTY>,RDDONE
        gettyp  r0,r1                   ; get flags
        setb    <#,F$LIST>,r1           ; make a list
        settyp  r1,r0
        movi    <#,0>,r1
;
; return

RDDONE: popm    r2,r3
        return
;
;extra right bracket error

RDERR1: movi    <#,E$ERBL>,r1
        jmp     RDERR
;
;missing right bracket error

RDERR2: movi    <#,E$MRBL>,r1
                                ;fall thru into RDERR
```

```
;
; failure

RDERR:  movp    <#,UNDEF>,r0            ; undefined signals error
        swapp   <#,NULL>,RDOOBJ         ; dealc. any partially read object
        jmp     RDDONE
;
; end of chars. encountered while reading word

RDEND:  jmppi   LEVEL,RDERR2            ; if sub-lists not complete, error
        movi    <#,0>,r0                ; 0 signals last char. was end
        jmp     READWD                  ; else return word

;
;[
;routine: GETPRF -- for input buffer initialization... hands BININI
;                   a pointer to the print-form argument to PARFRM.
;attr: SAFE
;]

        vsect

PRFPTR: ptr     null            ; a pointer to the print-form object.

        psect

GETPRF: movp    PRFPTR,r0
        movp    <@,0>,r0
        return

;[
;routine: PRFCHR -- supplies a print form's chars. to RDOBJ for PARFRM
;                   suplies opening & closing square brackets so that
;                   a line of text may be read in as a list
;output reg: r0
;globals: SWITCH (set up by READLN)
;attr: SAFE
;type: called by RDOBJ
;]
        vsect

SWITCH: int     0               ;0 means supply [
                                ;1 means get char. from file
                                ;-1 means ] line done
        psect

PRFCHR: jmpzi   SWITCH,OPNCHR
        jmpmi   SWITCH,NOCHR
        push    r1
        movp    <#,RDIBUF>,r1
        call    BINP
        pop     r1
        tstb    <#,B$DONE>,RDIFLG,CLSCHR
        succeed
;
; send [ to start list

OPNCHR: movi    <#,"[">,r0
        movi    <#,1>,SWITCH
        succeed
;
; send ] to end list

CLSCHR: movi    <#,"]">,r0
        movi    <#,-1>,SWITCH
        succeed
;
; signal EOL to caller

NOCHR:  movi    <#,0>,r0
        fail


;[
;routine: PARFRM (#PARSEFORM) - takes a "print form" string such as that
;          generated by #PRINTFORM and returns the LOGO object which is
;                    represented by that string.
;input regs: std. normal primitive
;output regs: r0
;attr: UNSAFE
;]
        entry   PARFRM

PARFRM: jmpep   <@,2>,<#,EMPTY>,REMPTY ; parse EMPTY as EMPTY
        gettyp  <@,2>,r0               ; check that arg is a word
        tstb    <#,F$LIST>,r0,PSFLIW    ; generate error otherwise
        movp    r2,PRFPTR
        movi    <#,B$STR>,RDIFLG        ; initialize input buff.
        movp    <#,GETPRF>,RDIOBJ
        movi    <#,1>,RDIOBC
        movp    <#,RDIBUF>,r1
        call    BININI
;
; call RDOBJ to do the work.

        movi    <#,0>,SWITCH            ; initialize bracket kludge
        movp    <#,PRFCHR>,rl           ; routine to do the reading
        call    RDOBJ
        jmpei   <#,UNDEF>,r0,PSFERR     ; check for error in RDOBJ
        succeed                         ; return object.
;
;  Error handling

PSFLIW: movi    <#,E$WELF>,rl           ; error if arg. is a list

PSFERR: movi    rl,ERRCOD
        jmp     STDERR                  ;standard error return

        endmod  READER
```

4.2.16 TRPHDL.LIL

```
       search LI1

        module TRPHDL -- LOGO trap handling routines

        include CONSTS
        include FLAGS
        include ERRORS

        extrn   $PI, CURPRC, PRVUSR, EMPTY,TRUEP, FALSP
        extrn   LEVEL
        extrn   ERRHDL, ERRARG, ERRCOD, ERROBJ, ERRINF, ERRCNT
        extrn   STDERR, GIVEUP, CMDDON, CRNUM, PRVUSR, CVTSTR, CVTNU0
        extrn   SBOBUF,SBOFLG,SBOOBP,SBIBUF,SBIFLG,SBIOBP,SBIOBC
        extrn   BININI,BOUINI,BINP,BOUP,CATLST

        vsect
;
;       Trap tables

        entry TPRENT,TEVENT,TFLENT,TALENT,TTRENT,TCLENT,TKBENT,TPNENT

TRPPRC:                             ; stacks of trap data block
                                    ; trapping procedures
TPRENT: ptr     NULL
TEVENT: ptr     NULL
TFLENT: ptr     NULL
TALENT: ptr     NULL
```

```
TTRENT: ptr     NULL
TCLENT: ptr     NULL
TKBENT: ptr     NULL
TPNENT: ptr     NULL

; stacks are linked lists with each entry having the following format:
;
;       ******************
;       *   prefix       *
;       ******************
;       *NINTS(always 1)*
;       ******************
;       * pointer to     *
;       * stack frame    *
;       * receiving trap*
;       ******************
;       * ptr. to symbol*
;       *  table entry  *
;       ******************
;       *    link ptr    *
;       ******************
;

;
;temporaries for trap data block manipulation

TRPFRM: ptr     null            ;trap frame level
TMPENT: ptr     null            ;trap procedure symbol table entry
TRPDAT: ptr     null            ;trap data block ptr.
TMPRNT: int     0               ;holds result of $RINT
;
;storage for SPRINGTRAP

SPTTYP: int     0               ; type of trap to be sprung
SPTOBJ: ptr     null    ; ptr to trap object to be passed to handler
SPTCOD: ptr     null    ; ptr to trap code to be passed to handler

;
;temporaries for timer routines

TMPINT: int     0

;
; temporaries for calculating trap line number

NLOCS:  int     0
NARGS:  int     0
NLINES: int     0
LINENO: int     0
```

```
        entry   TRPPRC

        psect

; dispatch table for special handling for particular trap types
; set at assembly-time

TRPSPC: ptr     PHDLER                  ;parser error handling
        ptr     EHDLER                  ;normal error handling
        ptr     EHDLER
        ptr     EHDLER
        ptr     EHDLER
        ptr     IHDLER                  ;interrupt handling
        ptr     IHDLER
        ptr     IHDLER
;
; table of ptrs to names of the trap types

        extrn   PARSTR,EVLSTR,FILSTR,ALCSTR,TRCSTR,CLKSTR,KBDSTR,PANSTR

TNAMES: ptr     PARSTR
        ptr     EVLSTR
        ptr     FILSTR
        ptr     ALCSTR
        ptr     TRCSTR
        ptr     CLKSTR
        ptr     KBDSTR
        ptr     PANSTR
;
;symbolic offsets to the args. to the trapping procedure
; dependent on the word size of the local machine

        const   TTYPE,  0               ;trap type
        const   TOBJ,   1*MWRDSZ        ;trap object
        const   TCNTXT, 2*MWRDSZ        ;trap context
        const   TLEVEL, 3*MWRDSZ        ;trap level
        const   TPROC,  4*MWRDSZ        ;trap procedure
        const   TLINE,  5*MWRDSZ        ;trap line (number)
        const   TCODE,  6*MWRDSZ        ;trap   code

;
;[
;routine TRPHDL: trap handler, TRPERR entry point for errors
;input reg: r1 — error type (TRPERR only)
;output reg: r0,r1 clobbered
;globals: $PI, CURPRC
;attr: SAFE
;type: called by evaluator or primitive, success return expects to
;       go to the start of the evaluator (EVAL)
;]
        entry   TRPHDL, TRPERR
;
; entry point for asynchronous interrupts

TRPHDL: movi    $PI, r1                 ;get code of interrupt
;
; entry point for error interrupts

TRPERR: pushm   r2,r3
        movi    <#,0>,$PI
        host    $RINT                   ;call $RINT on all traps
        movi    r0,TMPRNT               ;save result of RINT

        cvtip   r1,r3                   ;make ptr. for use as index
                                        ; into trap tables
        decp    r3                      ;(so $PI = 1 points to start of
                                        ; tables)
        jmpep   <#,NULL>,<TRPPRC,3>,TRPFAI ;fail if particular trap
                                        ; not enabled
        subvec  <TRPPRC,3>,<#,TRPFRM>,<#,2>,<#,2> ;get trap data
        jmpnep  TRPFRM,<#,NULL>,TRPFAI  ;fail if particular trap
                                        ; already activated
;
;trap can be sprung

        jmp     TRP0

; entry point for software-initiated trap

TRPSPR: pushm   r2,r3
        iori    <#,EV$SPT>,r1           ; mark frame as SPRINGTRAP frame
TRP0:   iori    r1,<EVFLGS,4>           ;mark stack frame as trap recipient
        movp    r4,TRPFRM               ;get pointer to current frame
        rplvec  <#,TRPFRM>,<TRPPRC,3>,<#,2> ;store frame pointer

;
;set up arguments to trapping procedure
;push args. onto LOGO stack

        movp    <TOP,4>,r2              ;r2 will be local stack ptr.
;
; get trap type

        swapp   <TNAMES,3>,<TTYPE,2>
```

```
;
; search thru stack frames for first named (not monitor or RUN) procedure

        movp    r4,r1
TRP1:   tstb    <#,EV$MON>,<EVFLGS,1>,TRPMON     ;monitor procedure found
        jmpep   <#,NULL>,<PROC,1>,TRP2A ;don't jump if
                                        ; named procedure found
        jmpnzi  PRVUSR,TRP2B            ;keep looking if priv. frame &
        tstb    <#,EV$PRV>,<EVFLGS,1>,TRP2A ;not priv. user
        jmp     TRP2B
TRP2A:  movp    <PREV,1>,r1
        jmp     TRP1
;
; get trap procedure name for named procedure

TRP2B:  getnam  <PROC,1>,r0
        movp    <#,NULL>,<TPROC,2>
        movs    r0,<TPROC,2>
;
; get line no.

        push    r3
        getval  <PROC,1>,r3             ;ptr. to proc. dir.
        subvec  r3,<#,NLOCS>,<#,2>,<#,3>;get NLOCS,NARGS,NLINES of proc
        movi    <LINE,1>,r0             ;offset to parse of current line
        subi    NLOCS,r0
        subi    NARGS,r0
        subi    NLINES,r0               ;r1 now has offset to line no.
        subvec  r3,<#,LINENO>,r0
        movi    <#,2>,r1
        call    CRNUM                   ;create number to hold trap level
        rplvec  <#,LINENO>,r0,<#,2>
        omovp   r0,<TLINE,2>
        pop     r3
        jmp     TRP3
;
; when trapped procedure is monitor, make :EMPTY proc name & line no.

TRPMON: movp    <#,EMPTY>,<TPROC,2>
        movp    <#,EMPTY>,<TLINE,2>
;
; set level, trap code, trap context

TRP3:   movi    <#,2>,r1
        call    CRNUM                   ;create number for trap level
        rplvec  <#,LEVEL>,r0,<#,2>
        omovp   r0,<TLEVEL,2>
        jmpep   <#,NULL>,SPTCOD,TRP3A   ; handle code if set by SPRINGTR
        cmovp   SPTCOD,<TCODE,2>
        movp    <#,NULL>,SPTCOD
        jmp     TRP6
TRP3A:  jmpzi   ERRCOD,TRP4
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,ERRCOD>,r0,<#,2>
        jmp     TRP5
TRP4:   movi    <#,1>,r1
        call    CRNUM
TRP5:   swapp   r0,<TCODE,2>
TRP6:   swapp   ERRINF,<TCNTXT,2>
;
;dispatch to special handling

        movp    TMPENT,CURPRC           ;give symbol table entry of
                                        ; trapping procedure to evaluator
        movp    <TRPSPC,3>,r0
        call    <@,0>                   ;call special handling routine
;
; if trap was sprung, ovrlay trap object with argument to SPRINGTRAP

        jmpep   <#,NULL>,SPTOBJ,TRPRET
        swapp   SPTOBJ,<TOBJ,2>
        movp    <#,NULL>,SPTOBJ
;
; Successful trap return

TRPRET: popm    r2,r3
        succeed
;
; Unsuccessful trap return

TRPFAI: host    $DINT                   ;r1 already set up
        popm    r2,r3
        fail

;
; special handling routines

;
;for parse errors--set trap object from error object

PHDLER: cmovp   ERROBJ,<TOBJ,2>
        return
;
;for normal errors, set trap object from integer in ERRCNT
; unless ERRCNT is 0...then trap object supplied
```

```
EHDLER: jmpzi   ERRCNT,PHDLER
        movi    <#,2>,rl
        call    CRNUM
        rplvec  <#,ERRCNT>,r0,<#,2>
        swapp   r0,<TOBJ,2>
        return
;
; for interrupts

IHDLER: movi    <#,2>,rl
        call    CRNUM
        rplvec  <#,TMPRNT>,r0,<#,2>     ;set trap object with $RINT value
        omovp   r0,<TOBJ,2>
        return
;
;trap handling utilities
;[
;routine:DISTRP  --dismisses a currently activated trap
;input reg: rl--trap channel no. (former $PI value)
;type: utility
;globals: $STATUS
;calls: $DINT
;]
        entry   DISTRP

DISTRP: push    r2
        cvtip   rl,r0                   ;get index to
        decp    r0                      ; trap tables
        subvec  <TRPPRC,0>,<#,TRPFRM>,<#,2> ;get pointer to frame in which trap
        movp    TRPFRM,r2               ; was received
        clrb    <#,EV$PI>,<EVFLGS,2>    ;unmark stack frame
        tstb    <#,EV$SPT>,<EVFLGS,2>,DISTRl  ; was trap soft-sprung?
        jmp     DISTR2          ; if so, don´t do $DINT
DISTRl: clrb    <#,EV$SPT>,<EVFLGS,2>
        jmp     DISTR3
DISTR2: host    $DINT                   ;signal dismissal of interrupt
DISTR3: movp    <#,NULL>,TRPFRM         ;mark trap as deactivated
        rplvec  <#,TRPFRM>,<TRPPRC,0>,<#,2>
        pop     r2
        return

;[
;ROUTINE:TRPTYP--gets trap entry corresponding to literal
;input REGS: r2--indirect ptr. to literal
;output REGS:r0--name table entry of literal,kills rl
;ATTR:SAFE
;TYPE:local utility
;GLOBAL:
        extrn SPT
;]

TRPTYP: gettyp  <@,2>,r0
        tstb    <#,F$LIST>,r0,TYPER1
        find    SPT,<@,2>,r0           ;look up literal
        jmpep   r0,<#,NULL>,TYPER2      ;jump if literal not found
        getflg  r0,r1
        tstb    <#,F$TRAP>,r1,TRPTY1    ;check if literal is trap type
        jmp     TYPER2

TRPTY1: succeed

TYPER1: movi    <#,E$WELF>,ERRCOD
        fail

TYPER2: movi    <#,E$UNTT>,ERRCOD       ;unknown trap type
        fail

;
;trap handling primitives

;[
;routine: SETTRP (SETTRAP)
;regs: std. normal primitive
;       1st arg. is trap name literal
;       2nd arg. is name of trap procedure
;globals:
        extrn   SPT,PT
;attr: UNSAFE
;]
        entry   SETTRP

SETTRP: call    TRPTYP,STDERR           ;get trap name table entry
        push    r3

SETTR1: decp    r2                      ;get 2nd arg.
        find    PT,<@,2>,r1            ;look up in procedure table
        jmpep   r1,<#,NULL>,TRPER2      ;jump if not found
        getflg  r1,r3
        tstb    <#,F$NTRP>,r3,TRPER2    ;proc. is wrong type to be trap
        andi    <#,F$ARGS>,r3
        jmpnei  r3,<#,7>,TRPER2         ;wrong number of args.
;
; create new trap data block

        movp    r1,TMPENT               ;store trap proc. table entry
```

```
      movi    <#,4>,r1
      movi    <#,F$VEC>,r2
      movp    <#,TRPDAT>,r3
      host    $ALLOC                  ;allocate trap data vector
      rplvec  <#,TMPENT>,TRPDAT,<#,3> ;store trapping proc. in data block
      getval  r0,r1                   ;get trap code
      cvtip   r1,r1
      decp    r1                      ;convert to index to trap tables
      movp    r1,r3
      addp    <#,TRPPRC>,r3           ;
      rplvec  r3,TRPDAT,<#,4>         ;store current trap data block
                                      ;as link ptr.
      omovp   TRPDAT,<TRPPRC,1>       ;put new trap data block in table
      pop     r3
      jmp     CMDDON
;
;trap handling primitive errors

TRPER2: movi  <#,E$PCBT>,ERRCOD       ;proc. can't be trap
      pop     r3
      jmp     STDERR
;[
;routine: POPTRP (PopTrap) -- removes current trap handler from
;     specified trap procedure stack and restoresthe old one.
;     the LOGO default handlers cannot be popped, nor can a
;     privileged handler by popped except by a privileged user.
;regs: std, primitive
;     1st arg iis trap type
;attr: SAFE
;]

      entry   POPTRP

POPTRP: call  TRPTYP,STDERR
      getval  r0,r1
      cvtip   r1,r0                   ; get index into trap tables
      decp    r0
      jmpep   <#,NULL>,<TRPPRC,0>,CMDDON  ;don't pop nothing
      tstb    <#,EV$PRV>,<EVFLGS,4>,POPTR2 ; check privileges on handler
      subvec  <TRPPRC,0>,<#,TMPENT>,<#,3>
      getflg  TMPENT,r1
      tstb    <#,F$PRIV>,r1,PPTER1    ; error if PRIV proc on stack
POPTR2: subvec <TRPPRC,0>,<#,TRPDAT>,<#,4> ; get next lower handler
      jmpep   <#,NULL>,TRPDAT,POPTR3
      subvec  <TRPPRC,0>,<#,TRPFRM>,<#,2> ;make sure trap isn't sprung
      jmpnep  TRPFRM,<#,NULL>,PPTER1  ; this is not a good error code.
      movi    <#,4>,TMPINT    ; set NINTS of trap stack entry to 4,
      rplvec  <#,TMPINT>,<TRPPRC,0>,<#,1>  ; so that the link ptr. to
```

```
                                ; the prev. frame won't get deallocated.
POPTR3: swapp   TRPDAT,<TRPPRC,0>       ; pop the old trap entry
        jmp     CMDDON

;errors:

PPTERl: movi    <#,E$OBPV>,ERRCOD
        jmp     STDERR

;
;
;[
;routine: SPRTRP (SpringTrap)--simulates the effect of the trap whose
;       type is the 1st arg. to this primitive, with a TrapObject value
;       of the 2nd arg, TrapContext of the 3rd arg, and TrapCode of 4th.
;input regs: std. primitive
;output: jumps back into evaluator (UNDEF is put on stack)
;calls: buffer management, UNARG, jumps to EVAL
;  JMPed to from the evaluator
;]

        entry   SPRTRP

        extrn   UNARG, EVAL, SPTSTR

SPRTRP: pushm   r1,r2
        movi    <#,3>,r1                ; convert string args
        call    CVTSTR,GIVEUP
        call    TRPTYP,SPRUTT           ; check trap type arg
        getval  r0,SPTTYP               ; save trap type index
        decp    r2
        swapp   <@,2>,SPTOBJ           ; and trap object
        decp    r2                      ; set trap context
        swapp   <@,2>,ERRINF
        decp    r2                      ; set trap code
        movi    <#,1>,r1
        call    CVTNU0,SPRNNF           ; convert to number
        swapp   <@,2>,SPTCOD           ; save for trap handler
        popm    r1,r2
        call    UNARG                   ; deallocate args to SPRINGTRAP
        movp    <TOP,4>,r1
        movp    <#,UNDEF>,<@,1>                 ; put UNDEF in result location
        movi    SPTTYP,r1
        push    r3
SPRTR1: cvtip   r1,r3                   ; get index into trap tables
        decp    r3
        jmpep   <#,NULL>,<TRPPRC,3>,SPRFAI      ; if trap not possible,
        subvec  <TRPPRC,3>,<#,TRPFRM>,<#,2>,<#,2> ; or if already set,
        jmpnep  TRPFRM,<#,NULL>,SPRFAI  ; generate a panic stop

;
;enter trap handler at appropriate point

        call    TRPSPR,SPRFAI
        pop     r3
        jmp     EVAL

;
; come here if trap attempt failed

SPRFAI: movi    <#,T$PANC>,r1
        jmp     SPRTR1

SPRNNF: movi    <#,5>,ERRCNT
        incp    r2                      ; restore args
        swapp   ERRINF,<@,2>
        incp    r2
        swapp   SPTOBJ,<@,2>
SPRERR: movp    <#,SPTSTR>,r3
        swapp   <#,NULL>,SPTOBJ
        popm    r1,r2
        jmp     TPRMER

SPRUTT: movi    <#,2>,ERRCNT
        jmp     SPRERR
;
;[
;routine: TRPSET (TrapSEt?) — returns TRUEP or FALSP as per whether a
;       given trap type is set or not.
;input regs: std. primitive, 1st and only arg is trap type.
;output: r0

        entry   TRPSET

TRPSET: call    TRPTYP,STDERR
        getval  r0,r0
        cvtip   r0,r1
        decp    r1
        jmpep   <#,NULL>,<TRPPRC,1>,TRPSTF  ; false if no trap handler
        subvec  <TRPPRC,1>,<#,TRPFRM>,<#,2> ; or if trap in progress
        jmpnep  <#,NULL>,TRPFRM,TRPSTF
        movp    <#,TRUEP>,r0
        succeed
TRPSTF: movp    <#,FALSP>,r0
        succeed
```

```
;[
;routine: TRPHLR (TrapHandler) -- returns list of all trap handlers on
;        the TRPPRC stack of the trap type given as arg.
;input regs: std. prim. (1st arg is trap type)
;output: r0 holds list of procedures
;attr: UNSAFE
;]

        entry   TRPHLR

TRPHLR: call    TRPTYP,STDERR
        push    r3
        getval  r0,r0
        cvtip   r0,r3
        decp    r3
        movp    <TRPPRC,3>,r3           ; get top level stack frame in r3
        jmpep   r3,<#,NULL>,TRPHLE              ; return empty if none
        movp    <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG        ; init. buffer management stuff
        movp    <#,SBOBUF>,r1
        call    BOUINI
        movp    <#,GETTPR>,SBIOBP       ; procedure to get trap name
;
; main loop thru stack frames

TRPHL1: subvec  r3,<#,TMPENT>,<#,3>     ; get symbol table entry for prc
        movi    <#,B$STR>,SBIFLG
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI
;
; character loop thru procedure name

TRPHL2: call    BINP
        tstb    <#,B$DONE>,SBIFLG,TRPHL3
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        movp    <#,SBIBUF>,r1
        jmp     TRPHL2
TRPHL3: subvec  r3,<#,TRPDAT>,<#,4>     ; get link to prev. handler
        jmpep   TRPDAT,<#,NULL>,TRPHLZ  ; if none, terminate main loop
        movp    <#,SBOBUF>,r1
        movi    <#,U$SPCS>,r2   ; otherwise output list separator
        call    BOUP
        movp    TRPDAT,r3
        jmp     TRPHL1
```

```
TRPHLE: movp    <#,EMPTY>,r0            ; return EMPTY if no handlers
        pop     r3
        succeed

TRPHLZ: pop     r3
        jmp     CATLST

;[
;routine: BAKTRC (Backtrace) -- returns list of active procedures on
;        the stack
;input regs: none (prim. with no args)
;output regs: r0 holds a pointer to a list of active procedure names
;calls: buffer management, gETTPR
;attr: UNSAFE
;]

        entry   BAKTRC

BAKTRC: push    r3
        movp    <#,EMPTY>,SBOOBP
        movi    <#,B$STR>,SBOFLG         ; init. buffer management stuff
        movp    <#,SBOBUF>,r1
        call    BOUINI
        movp    <#,GETTPR>,SBIOBP        ; procedure to get trap name
        movp    r4,r3                    ; r3 points to frame being
                                         ; looked at by BACKTRACE.
        movi    <#,0>,TMPINT    ; TMPINT flags whether a named frame
                                ; has been found yet.
BAKTR1: jmpep   <#,NULL>,<PROC,3>,BAKTR3         ; skip RUN frame
        movp    <PROC,3>,TMPENT          ; save procedure symtab entry
        jmpzi   TMPINT,BAKTR0            ; don't output separator
        movi    <#,U$SPCS>,r2
        movp    <#,SBOBUF>,r1            ; unless a name has been output
        call    BOUP
BAKTR0: incp    TMPINT
        movi    <#,B$STR>,SBIFLG
        movi    <#,1>,SBIOBC
        movp    <#,SBIBUF>,r1
        call    BININI
;
; character loop thru procedure name

BAKTR2: call    BINP
        tstb    <#,B$DONE>,SBIFLG,BAKTR3
        movi    r0,r2
        movp    <#,SBOBUF>,r1
        call    BOUP
        movp    <#,SBIBUF>,r1
        jmp     BAKTR2
BAKTR3: movp    <PREV,3>,r3             ; loop on next frame
        jmpnep  <$,STACK>,r3,BAKTR1

BAKTR4: pop     r3
        jmp     CATLST                  ; finish up

;[
; GETTPR — passes name of trapping procedure to BININI
;]

GETTPR: getnam  TMPENT,r0
        return

;

;[
;ROUTINE:TRAPRT (TrapReturn)—similar to VALUE, checks 1st arg.
;                                  for trap type
;a jump primitive (will jump back to evaluator)
;input REGS: std. primitive
; output reg:r1-stack frame flags,r3-parser entry point
;GLOBAL:ERRCOD
        extrn SPT
;CALLS:(jumps to,actually)
        extrn RUNPRC,PEXPR,UNFRM
;]
        vsect

FRMPTR: ptr NULL                        ;holds pointer to trapped frame
CONTXT: ptr NULL                        ;holds trap context string

        psect

        entry   TRAPRT

        extrn   TRTSTR                  ; needed for errors

TRAPRT: call    TRPTYP,TRTER1           ;r0 gets trap type table entry

TRAPR1: getval  r0,r1                   ;get trap code
        cvtip   r1,r0
        decp    r0                      ;r0 is index to trap tables
        jmpep   <$,NULL>,<TRPPRC,0>,TRTER2 ;no trap handler
        subvec  <TRPPRC,0>,<$,FRMPTR>,<$,2>
        jmpep   <$,NULL>,FRMPTR,TRTER2  ;trap not in progress
;
;unwind stack to trapped frame
```

```
        decp    r2                      ;point to trap context
        swapp   <@,2>,CONTXT           ;put trap context string on
                                        ; top of trapped frame's stack
TRAPR2: jmpep   r4,FRMPTR,TRAPR3
        call    UNFRM
        jmp     TRAPR2

TRAPR3: movp    <TOP,4>,r2              ;r2 points to CONTXT for VALOF
        swapp   CONTXT,<@,2>
        movi    <#,EV$TRT>,r1           ;mark frame as TRAPRETURN so it
        movp    <#,PEXPR>,r3            ; can handle UNDEFined results
        jmp     RUNPRC                  ;jump back to evaluator

TRTER1: movi    <#,E$UNIT>,ERRCOD
        jmp     TRTERR

TRTER2: movi    <#,E$TNIP>,ERRCOD
TRTERR: movi    <#,2>,ERRCNT
        movi    <#,2>,r1                ; restore arg count for ERRARG
        movp    <#,TRTSTR>,r3           ; set up procedure name
                                ; and fall through into TPRMER code...

;[
; TPRMER... code to handle errors in JMP-type trap primitives
;            expects r1,r2 to point to arg buff, r3 to have proc. name.
;]

TPRMER: pushm   r1,r2
        call    ERRARG,GIVEUP           ; generate error context
        swapp   r0,ERRINF
        popm    r1,r2
        call    UNARG
        jmp     ERRHDL
;
;[
;routine: SETTIM — SetTimer primitive
;inputs: std. prim. registers
;       1 numeric arg, which is the number of milliseconds to elapse
;       before an interrupt is generated.
;outputs: NONE
;attr: SAFE
;]

        entry   SETTIM

SETTIM: subvec  <@,2>,<#,TMPINT>,<#,1>
        jmpgi   TMPINT,<#,2>,TIMERR
        jmpnzi  TMPINT,SETTI0
        movi    <#,0>,r1
        jmp     SETTI1
SETTI0: gettyp  <@,2>,r0
        tstb    <#,F$SIGN>,r0,TIMERR    ; error if negative duration
        subvec  <@,2>,<#,TMPINT>,<#,2>
        movi    TMPINT,r1               ; call host routine if arg
SETTI1: host    $CLOCK                  ; is in range
        jmp     CMDDON

TIMERR: movi    <#,E$AOR>,ERRCOD
        jmp     STDERR

;[
;routine: CLRTIM — ClearTimer primitive
; input regs: ignored (no arguments)
; output regs — UNDEF in r0
;attr: SAFE
;]

        entry   CLRTIM

CLRTIM: host    $CCLOCK
        jmp     CMDDON

;[
;routine: RDTIM — Timer primitive, returns # of milliseconds remaining
;                 until an interrupt is generated by the timer.
;input regs: ignored
;output regs: r0 has address of number containing the time left.
;attr: UNSAFE
;]

        entry   RDTIM

RDTIM:  host    $RCLOCK
        jmpzi   r0,RDTIM0
        movi    r0,TMPINT
        movi    <#,2>,r1
        call    CRNUM
        rplvec  <#,TMPINT>,r0,<#,2>
        succeed
RDTIM0: movi    <#,1>,r1
        call    CRNUM
        succeed
        endmod  TRPHDL
```

5. LISTINGS OF THE LOGO-S ENVIRONMENT

5.1 Bootstrapping

The file listed in this section is READ, and then RUN, upon initializing LOGO. It causes global system names to be iniitalized, and the LOGO-S library procedures to be READ and DEFINEd. The local name of the file (BOOTS.LGO on TOPS-20) is stored in the system name #BOOTS.

5.1.1 BOOTS.LGO

```
[ !bootstrapper!
    #stuff notexttrap normal 0 7 1 tt tobj tctxt tlv tp tln tod
10 [longprint [error in bootstrap] :tctxt;
             if :tod = 122, print [Procedure does not parse]];
    settrap "evaluator "notexttrap;
    #setmon;
    close :#boots;
! set global system names -- Implementation dependent !
    "#save#directory <- "LOGO-SAVE-FILES-DIRECTORY≠;
     "PreserveKilledText <- char  16     !^P!;
     "ViewPreservedText  <- char  22     !^V!;
     "YankPreservedText  <- char  25     !^Y!;
     "UntypePhysicalLine <- char  21     !^U!;
     "Delete             <- char 127     !<RUBOUT>!;
     "WordDelete         <- char  23     !^W!;
     "BackDeleteLogicalLine <- char 2    !^B!;
     "NextWord           <- char  14     !^N!;
     "ForwardCharacter   <- char   6     !^F!;
     "RetypeLine         <- char  18     !^R!;
     "EndLogicalLine     <- char   5     !^E!;
     "GetKilledText      <- char   7     !^G!;
     "SearchForward      <- char  19     !^S!;
     "DeleteRegion       <- char   4     !^D!;
     "Pls                <- char  10     !<LF>!;
     "Bell               <- char  7      !<BEL>!;
     "quote#char         <- char  64     !  !;
     "l#p                <- char  40     !(!;
     "r#p                <- char  41     !)!;
     "l#b                <- char  91     ![!;
     "r#b                <- char  93     !]!;
    "EOL                 <- char  13     !<CR>!;
    "LF                 <- char  10     !<LF>!;
    "Space              <- char  32     !<SP>!;
    "Tab                <- char  9      !<TAB>!;
    "Suspend            <- char 26      !^Z!;

    make "w#s#chars longsentence :space :eol char 11 char 12 :tab :pls;
    "dispatch#chars <- longsentence
         :Delete :WordDelete :DeleteRegion :UntypePhysicalLine
         :BackDeleteLogicalLine :GetKilledText :PreserveKilledText
         :YankPreservedText :ViewPreservedText :RetypeLine
         :ForwardCharacter :NextWord :SearchForward :EndLogicalLine
         :Eol;

       "#preserve#buffer <- :empty;
       "#kill$buffer <- :empty;

       "editmode <- "false;
       "##edit#called <- "false;
       "#procname <- :empty;
       "lastline  <- :empty;

! protect names !

    make "x usernames;
    {until empty? :x,setproperty#name first :x "primitive; "x <- butfirst :x};
    setproperty#name "#save#directory "invisible;

! read in LOGO library (ies) !
    make "f2 "hacks.lgo; open :f2 "read;print [open hacks];
    make "go "true;{while :go do make "text read :f2;
    {if empty? :text then make "go "false else
    run sentence "#stuff :text}};print [open library];close :f2;
    make "f "library.lgo; open :f "read;make "go "true;
    {while :go do make "text read :f; {if empty? :text then make "go "false
    else run sentence "#stuff :text}};print [open spare];close :f;
    make "f "spare.lgo; open :f "read; make "go "true;
    {while :go do make "text read :f; {if empty? :text then make "go "false
    else run sentence "#stuff :text}};close :f;
    print [Throwing away text];make "x userprocs;
    {while not empty? :x do { if not property?#procedure first :x "primitive,
      notext first :x; setproperty#procedure first :x "primitive};
    "x <- butfirst :x};
    print [LOGO-S library read in]
]
```

5.2 The LOGO-S Library

5.2.1 HACKS.LGO

```
[userarrays normal 0 0 1
10      [return []         !  ***NOT YET IMPLEMENTED***  !]
]
[userfiles normal 0 0 2
10      [if word? :#save#directory,
         open "LOGO-SAVE-FILES-DIRECTORY≠ "read;
         "#save#directory <- read "LOGO-SAVE-FILES-DIRECTORY≠;
         close "LOGO-SAVE-FILES-DIRECTORY≠]
20      [return :#save#directory]
]
```

5.2.2 LIBRARY.LGO

```
[first#char normal 1 1 3 stringname char
10 [!extracts first character or quoted character pair from variable
     named by stringname!
     make "char first ::stringname;
     make ;stringname butfirst ::stringname ]
20 [ if quote#char? :char,
        make "char word :quote#char first ::stringname;
        make :stringname butfirst ::stringname ]
30 [return :char]
]
[last#char normal 1 1 3 stringname char
10 [!extracts last character or quoted character pair from variable
     named by stringname!
     make "char last ::stringname;
     make :stringname butlast ::stringname ]
20 [ if quote#char? last ::stringname,
        make "char word :quote#char :char;
        make :stringname butlast ::stringname ]
30 [return :char]
]
[read#char normal 1 0 2 char
```

```
10 [! reads in character or quoted pair from console !
      make "char readchar "console;
      if quote#char? :char,
       make "char word :quote#char readchar "console ]
20 [ return :char ]
]
[not#deleting normal 0 0 1
10 [return empty? :delete#string]
]
[delete#char normal 1 0 6  delchar
10 [ !deletes one character (or super-quoted pair) from output buffer!
     if both? (empty? :output#buffer) not#deleting,
      give#warning ]
20 [ make "delchar last#char "output#buffer ]
25 [ if :delchar = :l#b, make "bracket#count :bracket#count - 1
     else
     if :delchar = :r#b, make "bracket#count :bracket#count + 1 ]
30 [ if not#deleting, type "\ ]
40 [ make "delete#string word #echo :delchar :delete#string ]
50 [ if empty? :output#buffer,
      terminate#deletion; unwind :edit#level ]
]
[insert#char normal 0 1 4 chr
10 [ !inserts char or quoted pair into output buffer,displaying inserted chars!
      if :chr = :r#b,
       { if zero? :bracket#count, give#warning
            else make "bracket#count :bracket#count - 1}]
20 [ if :chr = :l#b, make "bracket#count :bracket#count + 1]
30 [ terminate#deletion ]
40 [ make "output#buffer word :output#buffer #echo :chr ]
]
[next#char normal 1 0 3 nexchr
10 [ !takes character (pair) from input buffer to output buffer!
     if empty? :input#buffer, give#warning; done ]
20 [ make "nexchr first#char "input#buffer ]
30 [ insert#char :nexchr ]
]
[update#kill#buffer normal 0 0 1
10 [ if (count :delete#string) > 1, make "#kill#buffer :delete#string ]
]
[terminate#deletion normal 0 0 4
10 [ if not#deleting, done ]
20 [ type "\ ]
30 [ update#kill#buffer ]
40 [ make "delete#string :empty ]
]
[#echo normal 0 1 1 echostring
10 [!may do something special someday, for now just outputs & returns input!
```

```
    writechar "console :echostring; return :echostring ]
]
[quote#char? normal 0 1 1 chr
10 [return :chr = :quote#char]
]
[delete#search normal 1 2 2 search#object search#list chr
10 [!strips chars from end of search#object until chr is in search#list
     returns length of search#object before found char,
     -1 means not found!
     until empty? :search#object,
       make "chr last#char "search#object;
       {if not zero? member :chr :search#list,
        return count :search#object}]
20 [return -1 !signal not found]
]
[insert#search normal 1 2 2 search#object search#chr chr
10 [!strips chars from front of search#object until chr is search#chr
     returns length of search#object after found char,
     -1 means not found!
     until empty? :search#object,
       make "chr first#char "search#object;
       {if  :chr = :search#chr,
        return count :search#object}]
20 [return -1 !signal not found]
]
[give#warning normal 0 0 1
10 [writechar "console :bell;unwind :edit#level]
]
[check#buffer normal 0 1 1 buffer
10 [if empty? :buffer, give#warning]
]
[retype#phys#line normal 1 1 4 line copy
10      [! This routine takes the last line entered and retypes it on
        a blank line on the user's terminal.  It is used to tell the
        user what he/she has after substantial edits of the line. !]
20      ["copy <- :empty]
30      [until either? (equivalent? last :line :pls) (empty? :line) then
                "copy <- word last :line :copy;
                "line <- butlast :line
        ]
40      [! We now have the procedure back to the last pls, which
        is the same thing as the current piece of the line. !
        writechar "console word :eol :copy
        ]
]
[word#delete normal 1 0 2 temp
10 [! Remove all characters up to start of first word!
          check#buffer :output#buffer;
```

```
            make "temp "false;! signals if we´ve started deleting a word!
            until empty? :output#buffer,
             make "chr last :output#buffer;
             {if :temp, !look for whitespaces!
                 {if not zero? member :chr :w#s#chars, done}
             else
              make "temp zero? member :chr :w#s#chars
             };
             delete#char ]
20 [ ! only come here if deleted all of output#buffer!
       terminate#deletion ]
]
[delete#region normal 1 0 4 temp
10 [! Input character to delete back to!
           make "chr read#char;
           if not zero? member :chr :dispatch#chars, give#warning ]
20 [        make "temp delete#search :output#buffer (longsentence :chr);
            if :temp < 0, give#warning ]
30 [        until (count :output#buffer) = :temp,
               delete#char ]
40 [ if empty? :output#buffer, terminate#deletion ]
]
[untype#phys#line normal 0 0 1
10 [    until (first :delete#string) = :pls,
          delete#char ]
]
[delete#line normal 0 0 1
10 [! Throw away output buffer!
      delete#char; go 10 !delete#char will unwind to editstring ]
]
[get#killed#text normal 1 0 2 temp
10 [! Insert #KILL#BUFFER!
           update#kill#buffer;
           check#buffer :#kill#buffer;
           make "temp :#kill#buffer ]
20 [  until empty? :temp,
            insert#char first#char "temp ]
]
[next#word normal 1 0 2 chr
10 [! get next word in input buffer!
     check#buffer :input#buffer;
     make "chr :empty ]
20 [ until not zero? member :chr :w#s#chars,
          {if empty? :input#buffer,
              terminate#deletion; done};
          make "chr first#char "input#buffer;
          insert#char :chr ]
]
```

```
[search#forward normal 2 0 5 chr temp
10 [! input character to search for in input#buffer
       insert up to & including that character!
     make "chr read#char ]
20 [ if :chr = :eol, end#logical#line; done ]
30 [ if not zero? member :chr :dispatch#chars, give#warning ]
40 [ make "temp insert#search :input#buffer :chr;
     if :temp < 0, give#warning ]
50 [ until (count :input#buffer) = :temp, next#char ]
]
[end#logical#line normal 0 0 1
10 [! Append rest of input#buffer to output#buffer!
     check#buffer :input#buffer;
     until empty? :input#buffer, next#char ]
]
[editstring normal 4 2 20 output#buffer input#buffer
        edit#level chr delete#string bracket#count
10       [! This routine allows the user to edit the text of a logical line
         on a hardcopy-oriented terminal.  It assumes that "line" is one
         large word, including any formatting characters the user enters,
         without the word containing the statement line number.  Therefore,
         the caller must find us the right object to edit.  If this routine
         was called as a dummy from REQUEST, notice that any attempt to
         print or delete the "rest of the procedure or line" will always
         ring the bell and do nothing.
         The characters which accomplish the edit are assumed to be predefined
         in BOOTS.]
20       [! "Output#buffer" is our copy of the logical line so far.!

         writechar "console :output#buffer;
         make "bracket#count 0;
         make "edit#level level;          !for unwinding to!
         make "delete#string :empty]

40       ["chr <- read#char !read in character (or quoted pair)]

50       [go 100 + member :chr :dispatch#chars]

100      [! :CHR is not in :DISPATCH#CHARS.The character is self-inserting!
         insert#char :chr;
         go 40
         ]
101      [! DELETE: Remove last character in OUTPUT#BUFFER!
            delete#char;
            go 40 ]
102      [! WORDDELETE:!
            word#delete; go 40 ]
103      [! DELETEREGION:!
```

```
           delete#region; go 40]
104     [! UNTYPEPHYSICALLINE:!
           untype#phys#line; go 40 ]
105     [! BACKDELETELOGICALLINE:!
           delete#line; go 40 ]
106     [! GETKILLEDTEXT:!
           get#killed#text; go 40 ]
107     [! PRESERVEKILLEDTEXT: preserve kill buffer!
           update#kill#buffer;
           make "#preserve#buffer :#kill#buffer; go 40 ]
108     [! YANKPRESERVEDTEXT: Insert #PRESERVE#BUFFER!
           insert#char :#preserve#buffer;
           go 40 ]
109     [! VIEWPRESERVEDTEXT: view preserve buffer, then retype last
           physical line!
           terminate#deletion;
           writechar "console longword :lf :#preserve#buffer :eol;
           retype#phys#line :output#buffer;
           go 40]
110     [! RETYPELINE: retype input & output buffers!
           terminate#deletion;
           writechar "console longword :eol :input#buffer :eol :output#buffer;
           go 40]
111     [! FORWARDCHARACTER: get next input buffer character!
           next#char; go 40]
112     [! NEXTWORD:!
           next#word; go 40]
113     [! SEARCHFORWARD:!
           search#forward; go 40]
114     [! ENDLOGICALLINE:!
           end#logical#line; go 40]
115     [! EOL: Return immediately, without copying anything else into the
           OutputBuffer.  However, do a check for unbalanced brackets !
           if  zero? :bracket#count,
            terminate#deletion;
            print :empty;
            return :output#buffer
           else
             give#warning; go 40]
]
[editline quoting 0 1 1 arg
10      [! This routine is a dummy for the EDL routine.  It merely passes
        its argument straight through to that routine. !
        edl :arg
        ]
]
[top#level greedy 0 1 1 message
10 [fullprint :message; unwind :toplevel]
```

```
]
[editmode#check normal 0 0 1
10 [ if not :editmode,
     top#level "You "are "not "editing.]
]
[real#insert#line normal 4 3 4 line text rflag save maybe newline nextline
10      [! This routine copies :#editbuffer, searching for the
        line number specified in :line.  If :text is empty?, the line
        is deleted if it existed.  Otherwise, :text replaces an existing
        line, else just adds the line
        (in the proper numerical order) to the procedure.
        :rflag signals message-printing (i.e,called by RESTORE#RANGE)!
         editmode#check
        ]
20      ["save <- butfirst :#editbuffer;
         "newline <- sentence :line :text;
        "#editbuffer <- longlist first :#editbuffer; ![[titleline]]!
        ]
30      [repeat count :save do !once for every line of the old procedure!
        "nextline <- first :save;
        "save <- butfirst :save;
                ! Get the line number we're at in the existing procedure !
        "maybe <- first :nextline;
                {if :line = :maybe then
                        ! We found an existing line with the same line
                        number:  replace it with our copy & exit!
                 {if not empty? :text,
                   make "#editbuffer lastput :#editbuffer :newline
                 };
                 make "#editbuffer sentence :#editbuffer :save;
                 {if :rflag,
                    fullprint longsentence "Line :line "replaced;
                 };
                 done
         else if :maybe > :line then
                        ! Line did not exist:  we are now looking at
                        the next higher line, so just insert ours before it
                        then exit!
                 {if not (empty? :text),
                   make "#editbuffer lastput :#editbuffer :newline
                 };
                 make "#editbuffer lastput :#editbuffer :nextline;
                 make "#editbuffer sentence :#editbuffer :save;
                 {if :rflag,
                     fullprint longsentence "Line :line "inserted;
                 };
                 done
               else
```

```
                ! Nothing special; we're not where we can tell
                  if the line exists or not, so merely recopy so far. !
                "#editbuffer <- lastput :#editbuffer :nextline
             }
        ]
40      [! New line goes at end of procedure. !
           "#editbuffer <- lastput :#editbuffer :newline;
           {if :rflag,
               fullprint longsentence "Line :line "inserted;
           }]
]
[insert#line normal 0 2 1 line text
10 [real#insert#line :line :text "false !no message printing]
]
[line#no? normal 0 1 1 no
10 [if  (integer? :no),
        return :no > 0
    else return "false ]
]
[range#pair? normal 3 1 5 pair#name temp first#value last#value
10 [!returns (thru :pair#name) canonical pair of line numbers
     returns false if ::pair#name can't be a range#pair!
    make "temp ::pair#name;
    if line#no? :temp, !single number, make into list!
      make :pair#name longsentence :temp;
      return "true ]
20 [ if not both? (flatlist? :temp) ((count :temp) = 2), return "false ]
40 [ make "first#value first :temp; make "last#value last :temp;
     if not both? (line#no? :first#value)(line#no? :last#value), return "false]
50 [ if :first#value > :last#value,
      make :pair#name sentence :last#value :first#value]
60 [ return "true]
]
[restore#range normal 1 1 3 range newtext
10       [! This routine restores a range of line numbers to their
         unedited states.  Its argument is assumed to be a list of
         two numbers (checked by RESTORE), the first being <= the
         second.
         It further assumes that the text of the routine is available,
         as this routine would never be called if we weren't already
         in editmode, and, if we are in editmode, the text must
         have been available. !]
20       [! Get the lines of the new text in range!
          make "newtext range#line text :#procname :range;
         if empty? :newtext,
          top#level "Nothing "to "restore.]
30       [! Now replace appropriate portion of :#editbuffer!
           until empty? :newtext,
            real#insert#line first first :newtext last first :newtext "true;
            make "newtext butfirst :newtext]
]
[range#line normal 2 2 3 proctext range inside line
10 [! returns a list of the portions of proctext (a text list)
      inside the line no. range specified by :range!
      make "proctext butfirst :proctext;
      make "inside :empty ]
20 [  until empty? :proctext,
       make "line first :proctext;
       {if lesseq? (first :range) (first :line),
        {if greater? (first :line) (last :range),
            return :inside
        };
         make "inside lastput :inside :line
        };
      make "proctext butfirst :proctext ]
30 [ return :inside ]
]
[restore quoting 1 1 6 x newtitle
10 [ !abort edit or partially restore changes!
     editmode#check ]
20 [ if range#pair? "x then !returns canonical pair in :x!
       ! We have a range of line numbers, as in SHOW.!
     restore#range :x; done ]
30 [  if equivalent? :#procname :x then
           ! Abort the edit of the entire procedure. !
           "editmode <- "false;
           top#level "Procedure :#procname "is "not word :#redefining "ed]
40 [ if not equivalent? "title :x then! nothing works!
      top#level :x "is "not "a "legal "argument ]

45 [! Restore the title line only. !
     make "newtitle titleline :#procname;
     if empty? :newtitle,
        top#level "No "previous "title ]
50 [ make "#editbuffer firstput :newtitle butfirst :#editbuffer;
     top#level "Restored "title ]
]
[end normal 0 0 1
10      [! This routine saves the copy of the routine that was being
        edited (in :#editbuffer).  It will object
        if it was not in EDIT mode when called.  It also resets from
        EDIT mode. !
        editmode#check;
        define :#editbuffer;
        "editmode <- "false;
        top#level "Procedure :#procname word :#redefining "ed ]
```

```
]
[edl normal 1 1 4 line text
10      [! This routine looks for the given line number (in "line") and
        calls editstring with the text of that line if it finds it.  If
        we were in the wrong mode, the line was illegal (more than one
        word, empty, or not an integer), or the line cannot be found,
        an appropriate diagnostic is generated.!
        editmode#check ]
20      [if not line#no? :line then
            top#level :line "is "not "a "line "number.]
30      [make "text last last range#line :#editbuffer (sentence :line :line);
         if empty? :text,
          top#level :line "not "found.]
40      [run editstring (word :line :space) :text ]
]
[#already#editing normal 0 0 1
10      [! Utility routine for EDIT and TO. !
        if :editmode,
         top#level "You "are "already (word :#redefining "ing) :#procname]
]
[text#check normal 0 1 1 procname
10      [if not procedure? :procname then
          top#level :procname "does "not "exist.
   else if primitive? :procname then
          top#level :procname "is "a "primitive.
   else if not text? :procname then
          top#level :procname "has "no "text.]
]
[edit quoting 0 1 2 procname
10      [! This routine sets up the rest of the system to do an edit.
        It sets the appropriate mode (and issues an error mode if it
        was not in the right mode to start an edit), puts an editable
        version of the procedure named in "procname" into the global
        variable :#editbuffer!
        #already#editing; text#check :procname]
20 [  "#editbuffer <- text :procname;
      "editmode <- "true;
      "#procname <- :procname;
      "#redefining <- "edit ]
]
[TO ewoc 0 1 1 tolist
10      [##to :tolist "TO]
]
[TOQUOTING ewoc 0 1 1 tolist
10      [##to :tolist "TOQUOTING]
]
[TOGREEDY ewoc 0 1 1 tolist
10      [if not (count element 2 :tolist) = 1 then
```

```
            top#level "You "must "have "exactly "one "argument "to "a "GREEDY.;
        else
                ##to :tolist "TOGREEDY]
]
[##TO normal 1 2 8 tolist proctype procname
10      [! Do some error checking. !
        if empty? :tolist then
                top#level :proctype "what?
        else if ((count :tolist) > 3),
                top#level element 4 :tolist "is "extra ]
12      ["procname <- first :tolist ]
13      [if list? :procname,
            top#level "Procedure "name "must "be "a "word]
14      [if not flatlist? element 3 :tolist,
            top#level "Arguments "must "be "a "flat "list]
15      [if not flatlist? element 4 :tolist,
            top#level "Locals "must "be "a "flat "list]

16      [if primitive? :procname,
               top#level :procname "is "a "primitive.]
20      ["#redefining <- "defin;
         if :##edit#called then !editing title line!
          "##edit#called <- "false;
          {if procedure? :procname,
           "#redefining <- "edit;
          }
         else !creating new procedure!
           #already#editing;
           {if procedure? :procname,
             top#level :procname "already "exists
           };
           make "editmode "true;
           make "#procname :procname;
           make "#editbuffer :empty ]
30      [ "#editbuffer <- firstput
                           (longlist :proctype :procname
                            element 2 :tolist element 3 :tolist)
                           butfirst :#editbuffer ]
]
[request normal 0 0 2
10      [if zero? column "console then
            type "*
        ]
20      [return #parseform editstring :empty :empty]
]
[define normal 9 1 6 proc name lines type args locals titleline temp temp2 line
10      [! This routine takes a procedure in the format the user
        knows a procedure as (i.e.  TO FOO [A B C] [D E F] ) and
        converts it to #STUFFable format.  It then #STUFFs the
        result.  If it finds that it was called with a procedure
        type that is not one of TO, TOGREEDY, or TOQUOTING, it
        issues an error message and unwinds to the current monitor.
        It is assumed that such a case could only be caused by a malfunction
        in a user routine or a direct user call to this routine,
        and thus should be terminated. !]
20      ["titleline <- first :proc]
30      [if equivalent? first :titleline "to then
                "type <- "normal
        else if equivalent? first :titleline "toquoting then
                "type <- "quoting
        else if equivalent? first :titleline "togreedy then
                "type <- "greedy
        else
                ! Unknown procedure type (108 is E$IMPF code)!
         springtrap "evaluator :type (sentence "DEFINE :proc) 108;
         unwind :toplevel
        ]
40      ["name   <- element 2 :titleline;
        "locals <- element 4 :titleline;
        "args   <- element 3 :titleline;
        "lines  <- butfirst :proc
        ]
50      [! Make lines into a flat list. !
        "temp <- [];
        "temp2 <- :lines;
        repeat count :lines do
                "line <- last first :temp2; !actual text!
                {if list? :line,
                    make "line no#brackets #textform :line
                };
                "temp  <- longsentence :temp first first :temp2
                                       #printform :line;
                "temp2 <- butfirst :temp2
        ]
60     [! We're done.  Just #STUFF it !
       #privrun longsentence "#stuff :name :type
                                    count :locals
                                    count :args
                                    count :lines
                                    :args :locals :temp
        ]
]
[#error normal 0 7 2 traptype trapobject trapcontext traplevel
                     trapprocedure trapline trapcode
10      [longtype :bell :space :traptype [#error detected. Code is:] :trapcode :eol;
        {if not empty? :trapcontext then
```

```
                type "Context#is#;fullprint :trapcontext
        };
        {if :traptype = "PARSE then
          type  [I had parsed up through#];fullprint :trapobject
          else if :trapobject > 1 then
                  longtype [Caused by arg. no.#] :trapobject - 1 :eol
        };
        {if not empty? :trapprocedure then
               longtype [I was at line#] :trapline [#in procedure#];
               fullprint :trapprocedure
        }
        ]
20      [unwind :toplevel       ! End of error handler definition. !]
]
[startup normal 0 1 1 initprompt
10 [    longtype :eol [LOGO-S test version#] version :eol :eol;

        ! Error trapping !

        settrap "keyboard "breaktrap;
        settrap "panic "abort;
        settrap "parse "errormessage;
        settrap "evaluator "errormessage;
        settrap "file "errormessage;
        settrap "store "#storetrap;
        monitor :initprompt ]
]
[monitor normal 1 1 1 prompt toplevel
10      [#setmon;
!        setproperty#name "prompt "primitive;
         setproperty#name "toplevel "primitive;
!       "toplevel <- level;
         while "true do runline
        ]
]
[runline normal 1 0 1 temp
10 [{if :editmode, type "e-};
    type :prompt;
    ! Store the last line typed at the current toplevel
    for use by LASTLINE. !
    make "ll#flag "true;!when true, lastline has been run!
    "temp <- editstring [] [];
    run :temp;
    if :ll#flag, "lastline <- :temp ]
]
[breaktrap normal 0 7 2 traptype trapobject trapcontext traplevel
                        trapprocedure trapline trapcode
5       [ print "Break;
```

```
          if not empty? :trapprocedure,
          longtype "Line# :trapline [#in procedure#];
          fullprint :trapprocedure
         ]
10       [suspend;
         if zero? :trapcode then
                 trapreturn :traptype :trapcontext
         else
         if thing? "trapcontext then
                 return :trapcontext
         else
                 trapreturn :traptype []
         ]
]
[#storetrap normal 0 7 3 traptype trapobject trapcontext traplevel
                        trapprocedure trapline trapcode
5        [ type [Storage overflow#];
          if not empty? :trapprocedure,
          longtype [in line#] :trapline [#in procedure#];
          fullprint :trapprocedure
         ]
7        [if :traplevel > :toplevel,
            print [probably due to too many active procedure calls];
            unwind :toplevel]
10       [print [of high severity--unwinding to topmost level];
          if zero? :trapcode then
                 trapreturn :traptype :trapcontext
         else
         if thing? "trapcontext then
                 return :trapcontext
         else
                 trapreturn :traptype []
         ]
]
[abort normal 0 7 2 traptype trapobject trapcontext traplevel
                    trapprocedure trapline trapcode
5        [ print [Panic stop];
          if not empty? :trapprocedure,
          longtype "Line# :trapline [#in procedure#];
          fullprint :trapprocedure
         ]
10       [unwind :toplevel]
]
[suspend normal 1 0 1 suspendlevel
10       [print "Suspending...;
         make "suspendlevel level;
         monitor word :prompt :initprompt
         ]
```

```
]
[continue normal 0 0 1
10 [if thing? "suspendlevel then
        unwind :suspendlevel
    else
         print [Can't continue further.]
   ]
]
[edt normal 0 0 2
10       [! This routine allows the user to edit the title line.
         When done, the (supposed) title line is passed back to
         ##to for evaluation.  A flag is set so that ##to does
         not object that the user was already in edit mode.
         Notice that this also allows someone to completely change
         the title line, say, to make it into an executable statement,
         and the line will be run.  Notice also that the first two
         words of the title are printed first, just as the line
         number of a line is printed first when the user starts
         a line edit. !
         editmode#check ]
20       ["##edit#called <- "true;
          ! Get the title after the name and type words !
           "temp <- butfirst butfirst first :#editbuffer;
           ! Now make one word out of it, without the outer
           list brackets !
           "temp <- butfirst butlast #printform :temp;
           run editstring (longword first first :#editbuffer :space
                           element 2 first :#editbuffer :space)
                           :temp ]
]
[no#brackets normal 0 1 1 word
10 [return butfirst butlast :word]
]
[show#lines normal 0 1 1 lines
10       [! This routine shows a sequence of lines, assumed to be
         stored as a list of sublists, each containing
         a line (i.e. [[10 foo] [20 bar] ... ])!
         until empty? :lines do
           print no#brackets #textform first :lines;
           make "lines butfirst :lines ]
]
[show#text normal 0 1 1 text
10 [! print procedure text in show format—
      blank line, title line, text lines!
         print [];
         print no#brackets #printform first :text;
         show#lines butfirst :text
         ]
```

```
]
[name normal 0 1 1 x
10      [! This routine takes a word or list of name names and
        returns the name names in a workspace descriptor
        list of the form [name [name-namel name-name 2 ...]].
        It does not check the names for existence; that is left
        to subordinate routines. !
        return list "name longsentence :x]
]
[procedure normal 0 1 1 x
10      [! This routine takes a word or list of procedure names and
        returns the procedure names in a workspace descriptor
        list of the form [procedure [proc-namel proc-name 2 ...]].
        It does not check the names for existence; that is left
        to subordinate routines. !
        return list "procedure longsentence :x]
]
[all quoting 0 1 3 x
10      [! This routine puts the names of all procedures, names, arrays,
        or files into the appropriate list structure. !]
20      [if list? :x then
                ! Not a valid option !
          top#level "ALL "must "be "followed "by "a "word ]
30      [if equivalent? :x "procedures then
                return list "procedure userprocs
        else if equivalent? :x "names then
                return list "name usernames
        else if equivalent? :x "arrays then
                return list "array userarrays
        else if equivalent? :x "files then
                return list "file userfiles
        else if equivalent? :x "titles then
                return usertitles
        else
                top#level "Wrong "keyword "for "ALL
        ]
]
[invisible quoting 0 1 3 x
10      [! This routine puts the names of all invisible
           procedures, names, or arrays into the appropriate list structure!]
20      [if list? :x then
                ! Not a valid option !
          top#level "INVISIBLE "must "be "followed "by "a "word ]
30      [if equivalent? :x "procedures then
                return list "procedure inv#procs
        else if equivalent? :x "names then
                return list "name inv#names
        else if equivalent? :x "arrays then
```

561

```
                return list "array :empty !for now!
        else
                top#level "Wrong "keyword "for "INVISIBLE
        ]
]
[primitive quoting 0 1 3 x
10      [! This routine puts the names of all primitive
           procedures, names, or arrays into the appropriate list structure!]
20      [if list? :x then
                ! Not a valid option !
          top#level "PRIMITIVE "must "be "followed "by "a "word ]
30      [if equivalent? :x "procedures then
                return list "procedure prv#procs
        else if equivalent? :x "names then
                return list "name prv#names
        else if equivalent? :x "arrays then
                return list "array :empty !for now!
        else
                top#level "Wrong "keyword "for "PRIMITIVE
        ]
]
[workspace normal 0 0 1
10      [! This routine takes all object names (except files) in the
        workspace and returns them in a classifier list!
        return longsentence
                        "procedure (longlist userprocs)
                        "name (longlist usernames)
                     !  "array (longlist userarrays)  !
        ]
]
[show#procedure normal 0 1 2 x
10      [! This routine shows the indicated procedure.  It also checks
        it for existence first, and complains if not found. !]
20      [if procedure? :x then
                show#text text :x
        else
                fullprint longsentence
                   "Procedure :x [does not exist]
        ]
]
[show#name normal 0 1 1 x
10      [! This routine shows the indicated name.  It also checks
        it for existence first, and complains if not found. !
        fulltype ": :x :space;
        if thing? :x,
           fulltype "<- :space ::x :eol
        else
         print [is undefined.] ]
```

562

```
]
[show greedy 1 1 6 x lines
10      [! This routine does the actual SHOWing of LOGO objects.
        See the Glossary for the types of input accepted. !
        "x <- first :x ]
20      [if empty? :x then
                ! Show the procedure as it is now edited, but not reDEFINEd. !
                editmode#check;
                show#lines :#editbuffer;
                done
        ]
30      [if range#pair? "x then
          editmode#check;
          make "lines range#line :#editbuffer :x;
          {if empty? :lines,
           {if (count :x) = 1,
                top#level "No "line first :x
            else
                top#level "No "lines "between first :x "and last :x
           }
          else
           show#lines :lines
          };
          done ]
40      [ if both? (list? :x) (list? first :x),
          ! title list !
          {until empty? :x,
           print butfirst butlast #textform first :x;
           make "x butfirst :x
          };
          done ]
50      [if arg#parse#setup "show "procedure :x then
            do#all#args;
            done ]
60      [top#level "Illegal "argument "to "SHOW]
]
[lastline normal 0 0 1
10      [! This routine allows the user to run the last line typed
           to the current monitor!
        make "ll#flag "false;
        run editstring :lastline []
        ]
]
[usertitles normal 2 0 5 x y
10      [! This routine returns a list of all user
        procedure title lines, each of which is itself a list
        (i.e.  [ [name1 type1 ... ] [name2 type2 ... ] ] ). !]
20      ["x <- userprocs]
```

```
30      ["y <- :empty]
40      [until empty? :x do
                "y <- lastput :y first text first :x;
                "x <- butfirst :x
        ]
50      [return :y]
]
[work#class#list? normal 0 1 3 x
10      [! This list returns "true if its argument is a legal workspace
        classifier list, and "false if it is not.  Both the list structure
        and the keywords in the list are checked for legality.
        A keyword with no objects after it (just an empty list)
        is perfectly legal; notice, however, that the empty list
        must still exist. !]
20      [until empty? :x do
                {if not word? first :x then
                        ! We have a list where we should have a keyword. !
                        return "false
                else if zero? member first :x
                [procedure procedures name names array arrays file files] then
                        ! Illegal keyword for workspace specifier !
                        return "false
                else if both? (word? element 2 :x)
                              (not empty? element 2 :x) then
                        ! The element after the keyword is not a list
                        of the names of objects. !
                        return "false
                else if both? (word? element 2 :x)
                              (not empty? element 2 :x) then
                        ! The element after the keyword is not a list
                        of the names of objects. !
                        return "false
                else
                        ! Okay so far; check the next part of the specifier. !
                        "x <- butfirst butfirst :x
                }
        ]
30      [return "true]
]
[work#title#list? normal 0 1 3 x
10      [! This routine takes its argument and returns "true if it
        is composed solely of lists (i.e.  there are no elements
        of the list that are words).  This must be assumed to
        be a list of title lines. !]
20      [until empty? :x do
                {if word? first :x then
                        ! Not only lists in the list !
                        return "false
                else
                        ! Okay so far. !
                        "x <- butfirst :x
                }
        ]
30      [return "true]
]
[titles normal 2 1 5 x y sofar
10      [! This routine returns a list of the titles of the
        procedures the user specifies.  Nonexistent routines
        are flagged, and the titles of only those that exist
        are returned.  Notice that, unlike PROCEDURE(S), etc.,
        the titles themselves are returned, not the routine names. !]
20      ["sofar <- :empty]
30      [if word? :x then
                "x <- longlist :x
        ]
40      [until empty? :x do
                "y <- text first :x;
                {if empty? :y then
                        longtype [Procedure#] #printform first :x
                                  [#does not exist.] :eol
                else
                        "sofar <- lastput :sofar first :y
                };
                "x <- butfirst :x
        ]
50      [if empty? :sofar then
                ! If we didn't find any procedures, return [procedure []]
                so that our superior can tell (i.e.  SHOW will
                thus not show the current procedure if in editmode),
                as it would if we just returned :empty. !
                return [procedure []]
        else
                ! Got some stuff to return; do so normally. !
                return :sofar
        ]
]
[title normal 0 1 1 x
10      [! This is a dummy for TITLES. !
        return titles :x
        ]
]
[titleline normal 0 1 1 x
10      [! This routine returns the title line of the indicated procedure!
           return first text :x]
]
[textline normal 0 2 2 x line
```

```
10      [! This routine returns the indicated line of the specified
        procedure.  It returns :empty if anything is wrong. !
        if not text? :x,
          return :empty ]
20      [ if not range#pair? "line,
              return :empty
          else
              return range#line text :x :line ]
]
[sats normal 0 0 1
10      [! This routine is an abbreviation for SHOW ALL TITLES. !
        show all titles
        ]
]
[replace normal 0 1 1 x
10      [! This routine allows the user to replace a line in a
        procedure. To do this efficiently, we just RUN the line. !
        run :x
        ]
]
[procedures normal 0 1 1 x
10      [! Dummy for PROCEDURE. !
        return procedure :x
        ]
]
[names normal 0 1 1 x
10      [! Dummy for NAME. !
        return name :x
        ]
]
[arg#parse#setup normal 2 3 9 action default argument quote numbersign
10      [! This routine take :argument and builds some global variables
        for arg#parse#dispatch to run.  In :##procedures, it puts the
        names of the procedures affected, in :##names, the names affected,
        and so forth.  In :##action, it puts "show#, "erase#, etc., so
        that all arg#parse#dispatch must do is concatenate the word
        in :##action with the type of the object it is manipulating
        ("procedure, "name, etc.), and call the procedure thus named.
        :default tells what type of objects we are to deal with
        if we get passed a flat list, and :action tells what
        to do with the object(s).  Thus, :default might be "name,
        and :action might be "erase, in a typical call.
        Notice that default is ignored if the list we were passed
        was not flat, and that any error causes the routine to
        return "false to its caller (else "true).  If an error
        occurs, the contents of the global variables in indeterminate. !]
20      ["quote <- char 34]
30      ["numbersign <- char 35]
```

```
40      ["##procedures <- :empty;
        "##names <- :empty;
        "##arrays <- :empty;
        "##files <- :empty;
        "##action <- :empty
        ]
50      [if word? :argument then
                ! We were passed a word to do whatever with; make
                it a list to allow easier processing. !
                "argument <- longlist :argument
        ]
60      [if flatlist? :argument then
                ! We were passed a flat list.  Use the default for
                the type of object. !
                {if zero? member :default
                [procedure procedures name names array arrays file files] then
                        ! Invalid default. !
                        longtype :default [#is not a valid default#]
                                  [to make a flat list into a workspace#]
                                  [classifier list.]
                                  :eol;
                        return "false
                else
                        ! Valid default. !
                        "argument <- (list :default :argument)
                }
        else if not work#class#list? :argument then
                ! Not a workspace classifier list:  something illegal. !
                fulltype :argument;
                print [#is not a flat list or a workspace descriptor list.];
                return "false
        ]
70      [! Now build the runnable objects for ARG#PARSE#DISPATCH. !
        until empty? :argument do
                "routinename <- longword :action :numbersign
                                         first :argument;
                {if procedure? :routinename then
                        ! We can call this one. !
                        {if not arg#parse#dump :argument then
                                ! The appropriate global variables
                                have been not set up because the
                                type of the object to be manipulated
                                in some part of what we passed was not
                                right; this is theoretically
                                impossible, but we check anyway.  The
                                error was printed by arg#parse#dump. !
                                return "false
                        }
                else
                        ! Illegal somehow. !
                        fulltype :routinename;
                        print [#does not exist to be called.];
                        return "false
                };
                "argument <- butfirst butfirst :argument
        ]
80      ["##action <- word :action :numbersign]
90      [return "true]
]
[arg#parse#dispatch normal 1 0 4 quote
10      [! This routine takes the global variable that ARG#PARSE#SETUP
        set up and merely runs the variable, piece by piece, returning
        after every object processed.  If it was called and an
        object existed for it to do something with, it returns "true,
        else "false. !]
20      ["quote <- char 34]
30      [if not empty? :##procedures then
                run sentence word :##action "procedure
                             word :quote first :##procedures;
                "##procedures <- butfirst :##procedures
        else if not empty? :##names then
                run sentence word :##action "name
                             word :quote first :##names;
                "##names <- butfirst :##names
        else if not empty? :##arrays then
                run sentence word :##action "array
                             word :quote first :##arrays;
                "##arrays <- butfirst :##arrays
        else if not empty? :##files then
                run sentence word :##action "file
                             word :quote first :##files;
                "##files <- butfirst :##files
        else
                ! Nothing left to run. !
                return "false
        ]
40      [return "true       ! Last call ran something. !]
]
[do#all#args normal 0 0 1
10      [! Utility routine for SHOW, etc. !
        while arg#parse#dispatch do
                {}
        ]
]
[arg#parse#dump normal 2 1 5 argument temp routines
10      [! This routine takes what was handed to arg#parse#setup as
```

```
        :argument, and puts the routines in it into the appropriate
        global variables.  If it finds an illegal object type
        (something which, given program logic, theoretically can't happen),
        it screams and returns "false, else "true. !]
20      [! Figure out what type of global variable the
        name belongs in, and put it there. !
        "temp <- first :argument
        ]
30      [! Store the object names in an easily accessible way. !
        "routines <- first butfirst :argument
        ]
40      [until empty? :routines do
                {if either? (equivalent? :temp "procedure)
                        (equivalent? :temp "procedures) then
                    "##procedures <- sentence :##procedures first :routines
                else if either? (equivalent? :temp "name)
                          (equivalent? :temp "names) then
                    "##names <- sentence :##names first :routines
                else if either? (equivalent? :temp "array)
                          (equivalent? :temp "arrays) then
                    "##arrays <- sentence :##arrays first :routines
                else if either? (equivalent? :temp "file)
                          (equivalent? :temp "files) then
                    "##files <- sentence :##files first :routines
                else
                    ! Theoretically, we should never get
                    here, because all object type names
                    have been checked previously.  However,
                    if we're here, tell people. !
                    longtype :temp [#is not a valid object type.]
                          :eol;
                    return "false
                };
                "routines <- butfirst :routines
        ]
50      [return "true]
]
[lines normal 2 1 8 x y sofar
10      [! This routine returns a list of all the line numbers
        of the specified procedure. !]
20      ["y <- text :x]
30      [if empty? :y then
                ! Routine does not exist. !
                longtype [Procedure#] #printform :x [#does not exist.] :eol
                return :empty
        ]
40      ["y <- butfirst :y]
50      [if empty? :y then
```

```
                ! Routine has no text. !
                longtype [Procedure#] #printform :x [#has no text.] :eol;
                return :empty
        ]
60      ["sofar <- :empty]
70      [until empty? :y do
                "sofar <- sentence :sofar first first :y;
                "y <- butfirst :y
        ]
80      [return :sofar]
]
[eds normal 0 2 1 x y
10      [! This routine is a synonym for EDITSTRING. !
        return editstring :x :y
        ]
]
[ed quoting 0 1 1 x
10      [! This routine is a synonym for EDIT. !
        run lastput [edit] :x
        ]
]
```

5.2.3 SPARE.LGO

```
[print#titles normal 1 1 2 y x
5       [! The argument is a file to be opened for output. !]
10      [open :y "write;
        "x <- userprocs;
        {while not empty? :x do
                writechar :y word #printform first text first :x :eol;
                "x <- butfirst :x
        }
        ]
]
[readin normal 1 1 3 x xx
5       [open :x "read]
10      ["xx <- "a]
20      [until empty? :xx do
                "xx <- read :x;
                #privrun sentence "#stuff :xx
        ]
]
```

```
[Get greedy 16 1 6 arg filename file wdl restrictions nname temp
    Want#procedures Want#names Want#commands procedures#to#get names#to#get
    procedures#skipped names#skipped procedures#defined names#defined
    at#top#level
10 [if not parse#arg#for#Get, ! determine legitimacy of arguments and
                                  then set up lists of items to get !
      fullprint [Get can't continue]; unwind :toplevel]
20 ["procedures#skipped <- []; "names#skipped <- [];
     "procedures#defined <- []; "names#defined <- [];
    open :file "read]
30 [until endfile? :file, "temp <- read :file;
     {if :at#top#level, type ".};  ! twiddle thumbs so that user
                                       doesn't worry that the system died !
     {if not zero? member (first first :temp) [to togreedy toquoting],
         {if :Want#procedures, get#procedure}
     else if "make = first :temp, {if :Want#names,
                                   "nname <- first butfirst :temp; get#name}
     else if "<- = #textform first butfirst :temp, {if :Want#names,
                                          "nname <- first :temp; get#name}
     else if not empty? :temp,
         {if :Want#commands, run :temp}}]
40 [close :file; if :at#top#level, print []]
50 [query#overwrite ! should existing items be overwritten? !]
60 [if :at#top#level,
      report#gotten "procedures; report#gotten "names;
      fullprint sentence :filename "completed.
      ! but keep silent if called from within a procedure !]
]

[Parse#arg#for#Get normal 0 0 12
10 ["filename <- first :arg]
20 [if list? :filename,           ! no args or list as first arg !
      fullprint [Get expects a SaveFile name as its first input];
      return "false]
30 [if 1 = count :arg,            ! filename only !
      "wdl <- [];
      "restrictions <- [] ]
40 [if 2 = count :arg,            ! second arg is either !
      "wdl <- first butfirst :arg;      ! a WorkspaceDescriptorList !
      {if "not = first :wdl,            ! or a list of restrictions !
          "restrictions <- :wdl;
          "wdl <- []
       else "restrictions <- []} ]
50 [if 3 = count :arg,            ! if 3 args, only one may be !
      "wdl <- first butfirst :arg;      ! a wdl and only one may be !
      "restrictions <- last :arg;       ! restrictions !
      "temp <- ("not = first :wdl);
    {if equivalent? :temp ("not = first :restrictions),
        {if :temp,
             fullprint [two sets of restrictions to Get]
         else fullprint [two lists of workspace constituents to Get]};
        return "false};
    {if :temp,                          ! name them correctly!
        "temp <- :wdl;
        "wdl <- :restrictions;
        "restrictions <- :temp} ]
60 [if 3 < count :arg,                  ! too many args !
    print [Get expects one to three inputs, but found];
    fullprint :arg;
    return "false]
70 ["Want#procedures <- not either? not zero? member "procedure :restrictions
                         not zero? member "procedures :restrictions;
   "Want#names <- not either? not zero? member "name :restrictions
                         not zero? member "names :restrictions;
   "Want#commands <- not either? not zero? member "command :restrictions
                           not zero? member "commands :restrictions]
80 [if empty? :wdl,   ! not providing names of specific items to get !
      "procedures#to#get <- []; "names#to#get <- []
   else if arg#parse#setup "get "procedure :wdl, ! wdl is legal !
      "procedures#to#get <- :##procedures ; "names#to#get <- :##names
   else return "false                           ! wdl is illegal !]
90 [if conflicts? ! between wdl and restrictions ! , return "false]
100 [if not savefile? :filename,        ! can't get !
      fullprint sentence :filename [is not a save file.]; return "false]
110 ["file <- longword :filename ".lgo-sav :space;
   "at#top#level <- zero? ( level - :toplevel - 4 ) ]
120 ["temp <- accesstype :file;
   if :temp = "unopened, return "true
   else print longsentence "file :file
                           word :l#p "named word :filename :r#p
                           [is already open for] :temp
                           [access.  Close] :file
                           [before continuing.]; return "false]
]

[Conflicts? normal 0 0 3
10 [if not either? :Want#procedures empty? :procedures#to#get,
    fullprint [You have specified procedures to get, yet
                   requested me to skip all procedures?];
    return "true]
20 [if not either? :Want#names empty? :names#to#get,
    fullprint [You have specified names to get, yet
                   requested me to skip all names?];
    return "true]
30 [return "false]
```

```
]

[Get#procedure normal 1 0 4 pname
10 ["pname <- first butfirst first :temp]
20 [if both? (not empty? :procedures#to#get)
             (zero? member :pname :procedures#to#get), done]
30 [if procedure? :pname,
       "procedures#skipped <- sentence :procedures#skipped list :pname :temp;
       done]
40 [define :temp; "procedures#defined <- sentence :procedures#defined :pname]
]

[Get#name normal 0 0 4
10 [ ! nname has already been determined by Get ! ]
20 [if both? (not empty? :names#to#get)
             (zero? member :nname :names#to#get), done]
30 [if thing? :nname,
       "names#skipped <- sentence :names#skipped list :nname :temp;
       done]
40 [run :temp; "names#defined <- sentence :names#defined :nname]
]

[query#overwrite normal 0 0 2
10 [if not empty? :procedures#skipped, Qover#procedures]
20 [if not empty? :names#skipped, Qover#names]
]

[member? normal 0 2 1 item object
10      [return not zero? member :item :object]
]
[butelt normal 0 2 5 elt obj
10  [if :elt < 1, return :obj]
20  [if :elt = 1, return butfirst :obj]
30  [if :elt > count :obj, return :obj]
40  [if :elt = count :obj, return butlast :obj]
50  [if word? :obj, return word subobject 1 :elt - 1 :obj
                                subobject :elt + 1 count :obj :obj
     else           return sentence subobject 1 :elt - 1 :obj
                                    subobject :elt + 1 count :obj :obj]
]

[Qover#procedures normal 5 0 8 locallist n pname location problem
5 [! locallist contains pairs of pname and procedure text for all
      skipped procedures !]
10 ["locallist <- :procedures#skipped; "n <- 1; type "procedures:##]
20 [until empty? :locallist,  ! list all skipped procedures with numbers !
          type longword :l#p :n :r#p "#; "n <- :n + 1;
          fulltype first :locallist; type word char 44 "#;
          "locallist <- butfirst butfirst :locallist]
30 [print [skipped.];
     print [Type list of numbers of procedures to redefine.]]
40 ["n <- request; ! get list of numbers corresponding to desired procedures !
    "locallist <- :n;
    until empty? :locallist,
          {if (count :procedures#skipped) < (2 * first :localllist),
              fullprint longsentence [There is no procedure numbered]
                                     first :locallist
                                     [in the list I gave you];
              go 10};
          "locallist <- butfirst :locallist]
50 ["locallist <- :procedures#skipped  ! reset locallist !]
60 [until empty? :n,
          "location <-  2 * first :n;
          "pname <- list element (:location - 1) :procedures#skipped
                         element :location :procedures#skipped;
          "location <- member :pname :locallist;
          "locallist <- butelt :location butelt :location :locallist;
          "n <- butfirst :n;
          "problem <- Gettable? "Procedure first :pname;
          {if empty? :problem,
              define last :pname;
              "procedures#defined <- sentence :procedures#defined first :pname
           else fullprint longsentence #printform first :pname
                                       [has property] :problem
                                       [and cannot be redefined.]}]
70 ["procedures#skipped <- :locallist]
]

[Gettable? normal 1 2 4 objecttype objectname temp
10 ["temp <- sentence (longword "property? "# :objecttype)
                       word char 34 :objectname]
20 [if value sentence :temp word char 34 "Primitive,
        return "Primitive]
30 [if value sentence :temp word char 34 "NoWrite,
        return "NoWrite]
40 [return []]
]

[Qover#names normal 5 0 8 locallist n nname location problem
5 [! locallist contains pairs of nname and value for all
      skipped names !]
10 ["locallist <- :names#skipped; "n <- 1; type "names:##]
20 [until empty? :locallist,  ! list all skipped names with numbers !
          type longword :l#p :n :r#p "#; "n <- :n + 1;
          fulltype first :locallist; type word char 44 "#;
          "locallist <- butfirst butfirst :locallist]
```

```
30 [print [skipped.]; print [Type list of numbers of names to rebind.]]
40 ["n <- request; ! get list of numbers corresponding to desired names !
    "locallist <- :n;
    until empty? :locallist,
          {if (count :names#skipped) < (2 * first :locallist),
              fullprint longsentence [There is no name numbered]
                                     first :locallist
                                     [in the list I gave you];
              go 10};
          "locallist <- butfirst :locallist]
50 ["locallist <- :names#skipped  ! reset locallist !]
60 [until empty? :n,
          "location <-  2 * first :n;
          "nname <- list element (:location - 1) :names#skipped
                         element :location :names#skipped;
          "location <- member :nname :locallist;
          "locallist <- butelt :location butelt :location :locallist;
          "n <- butfirst :n;
          "problem <- gettable? "name first :nname;
          {if empty? :problem,
              run last :nname;
              "names#defined <- sentence :names#defined first :nname
           else fullprint longsentence #printform first :nname
                                        [has property] :problem
                                        [and cannot be rebound.]}]
70 ["names#skipped <- :locallist]
]

[report#gotten normal 1 1 5 type temp
10 ["temp <- thing word :type "#defined]
20 [if empty? :temp, fullprint longsentence "No :type "defined.; done]
30 [type char 91]
40 [until 1 = count :temp, type #textform first :temp;
                           type word char 44 "#;
                           "temp <- butfirst :temp]
50 [print sentence #textform first :temp word "defined. char 93]
]

[Save normal 4 2 5 FileName Items saved? report file property#assignments
10 [if word? :#save#directory,
         open "LOGO-SAVE-FILES-DIRECTORY≠ "read;
         "#save#directory <- read "LOGO-SAVE-FILES-DIRECTORY≠;
         close "LOGO-SAVE-FILES-DIRECTORY≠]
20 ["file <- longword :filename ".lgo-sav :space  ;
     "property#assignments <- [];
     "report <- zero? ( level - :toplevel - 3 ) ]
30 [if not arg#parse#setup "Save "procedure :items, done]
40 [if not Savefile? :filename, create#savefile#entry :filename;
```

```
        Save#it; done
    else print longsentence [A file named] :filename
        [already exists.  Replace that file? ( type "yes" or "no" )]]
50 [if "yes = first request, print "proceeding... ; Save#it
    else print [you are unsaved]; done]
]

[Save#it normal 0 0 4
10 [open :file "write; "saved? <- "false]
20 [do#all#args]
30 [until empty? :property#assignments,
         fullwrite :file first :property#assignments;
         "property#assignments <- butfirst :property#assignments]
40 [if :saved?,
        {if :report, print sentence [saved on file] :filename};
        close :file
    else {if :report, print sentence [nothing saved on file] :filename};
        unopen :file]
]

[Save#Procedure normal 0 1 7 Pname
10 [if not procedure? :pname,
    fullprint sentence [Can't find procedure named] :pname; done]
20 [if not text? :pname,
    fullprint sentence :pname [has no text and cannot be saved.]; done]
30 [if property?#procedure :pname "Primitive,
    fullprint sentence :pname [is primitive and cannot be saved.]; done]
40 [fullwrite :file text :pname]
50 [if :report, fullprint sentence :pname "saved.]
60 ["property#assignments <-
        sentence :property#assignments ProcPropAssignments :pname]
70 ["Saved? <- "True]
]

[ProcPropAssignments normal 1 1 6 Pname Proplist
10 ["Proplist <- []]
20 [if property?#procedure :pname "Invisible,
      "Proplist <- lastput :proplist longlist
                                    "SetProperty#Procedure
                                    word "" :pname
                                    word "" "Invisible]
30 [if property?#procedure :pname "Traced,
      "Proplist <- lastput :proplist longlist
                                    "SetProperty#Procedure
                                    word "" :pname
                                    word "" "Traced]
40 [if property?#procedure :pname "Stepped,
      "Proplist <- lastput :proplist longlist
```

```
                                       "SetProperty#Procedure
                                       word "" :pname
                                       word "" "Stepped]
50 [if property?#procedure :pname "NoWrite,
        "Proplist <- lastput :proplist longlist
                                       "SetProperty#Procedure
                                       word "" :pname
                                       word "" "NoWrite]
60 [return :proplist]
]

[NamePropAssignments normal 1 1 4 Nname Proplist
10 ["Proplist <- []]
20 [if property?#name :nname "Invisible,
        "Proplist <- lastput :proplist longlist
                                       "SetProperty#Name
                                       word "" :nname
                                       word "" "Invisible]
30 [if property?#Name :nname "Traced,
        "Proplist <- lastput :proplist longlist
                                       "SetProperty#Name
                                       word "" :nname
                                       word "" "Traced]
40 [return :proplist]
]

[Save#name normal 1 1 9 Nname LocalVal
10 [if not thing? :nname,
    fullprint sentence :nname [has no value]; done]
20 [if property?#name :nname "NoRead,
    fullprint sentence :nname [has "NoRead property -- cannot be saved]; done]
30 [if property?#name :nname "Primitive,
    fullprint sentence :nname [is primitive and cannot be saved.]; done]
40 ["LocalVal <- ::Nname]
50 [if not list? :LocalVal, "LocalVal <- word "" :LocalVal]
60 [fullwrite :file longlist "make word "" :nname :LocalVal]
70 [if :report, fullprint sentence word ": :nname "saved.]
80 ["property#assignments <-
          sentence :property#assignments NamePropAssignments :nname]
90 ["Saved? <- "True]
]

[Savefile? normal 0 1 2 filename
10 [if word? :#save#directory,
         open "LOGO-SAVE-FILES-DIRECTORY7 "read;
         "#save#directory <- read "LOGO-SAVE-FILES-DIRECTORY7;
         close "LOGO-SAVE-FILES-DIRECTORY7]
20 [return not zero? member :filename :#save#directory]
```

```
]

[Create#Savefile#Entry normal 0 1 2 savename
10 ["#save#directory <- sentence :#save#directory :savename]
20 [open "LOGO-SAVE-FILES-DIRECTORYɏ "write;
        fullwrite "LOGO-SAVE-FILES-DIRECTORYɏ :#save#directory;
        close "LOGO-SAVE-FILES-DIRECTORYɏ]
]

[ErrorMessage normal 4 7 12 traptype trapobject trapcontext
                              traplevel trapprocedure trapline trapcode
                    #file #line #message #temp
10 [if :trapcode < 100, make "#line :trapcode
    else make "#line 1 + butfirst :trapcode]
20 [make "#file longword "<xlogo> :traptype ".trap :space]
30 [make "#temp accesstype :#file]
40 [if not :#temp = "unopened,
    longtype :#file [open with] :#temp [access.] :eol;
    print [Check, close, and continue.];
    suspend]
50 [open :#file "read]
60 [repeat :#line, make "#message read :#file]
70 [make "#message Error#Parse :traptype :trapobject
                              :trapcontext last :#message]
80 [if not empty? :trapcontext,
      type "Context#is:##; print :trapcontext]
90 [print #textform :#message]
100 [if not empty? :trapprocedure,
      print longsentence
      [at level] :traplevel
      [line] :trapline
      [in procedure] :trapprocedure]
110 [close :#file]
120 [unwind :toplevel]]
[Error#Parse normal 4 4 5 traptype trapobject trapcontext trapmessage
                    #agent #patient #advice #location
10 [if :traptype = "PARSE,
      make "#advice sentence [I parsed through:  ] :trapobject;
      go 50]
20 [if not integer? :trapobject, go 50]
30 [make "#patient element :trapobject :trapcontext;
     make "#agent first :trapcontext;
     make "#location ordinal :trapobject - 1]
40 [if list? :#patient, make "#patient longlist :#patient]
50 [return value sentence "longsentence :trapmessage]]
[ordinal normal 0 1 3 n
10 [if :n < 4, return element :n [first second third]]
20 [if 1 = last butlast :n, return word :n "th]
```

```
30 [return word :n element ( 1 + member last :n [1 2 3] ) [th st nd rd]]]
```

5.3 The Error Message Templates

The files listed in this section are OPENed and READ whenever the standard error trap handlers (defined in the LOGO-S library) are invoked. They contain RUNnable templates for the standard system error messages.

5.3.1 EVALUATOR.TRAP

```
[100 [:#agent [expects a word as its] :#location [input, but found] :#patient]]
[101 [:#patient [has no value]]]
[102 [:#agent [expects TRUE or FALSE as its] :#location [input, but found] :#patient]]
[103 [:#agent [expects a number as its] :#location [input, but found] :#patient]]
[104 [[You don't say what to do with] :trapcontext]]
[105 [:trapcontext [cannot be an input.  It does not return a value.]]]
[106 [[wrong number of arguments to] :#agent]]
[107 [:#agent [expects a list as its] :#location [input, but found] :#patient]]
[108 [[improper format for a special]]]
[109 [[division by 0 is illegal]]]
[110 [:#patient [is the wrong size object for] :#agent]]
[111 [[The input] :#patient [is out of range for] :#agent]]
[112 [:#patient [is not a valid trap type for] :#agent]]
[113 [:#patient [can't be a traphandler -- Primitive? Inputs?]]]
[114 [:#patient [is not a valid line number]]]
[115 [[Can't execute] :#agent [— not in a procedure]]]
[116 [[procedure] :trapprocedure [has no line numbered] :#patient]]
[117 [[Can't execute] :#agent [—] :#patient [trap not in progress]]]
[118 [:#agent [expects a non-numeric word as its]
          :#location [input, but found] :#patient]]
[119 [:#patient [does not exist]]]
[120 [:#patient [is protected — check its properties]]]
[121 [:#patient [is a non-existent property]]]
[122 [[text of] :#patient [can't be thrown away — it didn't parse]]]
[123 [[can't erase procedure] :#patient [—] :#patient [is a trap handler]]]
[124 [[can't erase procedure] :#patient [—] :#patient [is still active]]]
```

5.3.2 FILE.TRAP

```
[200 [:#patient [is not a valid access type for OPEN]]]
[201 [[file] :#patient [is already open]]]
[202 [[Can't] :#agent [yet — file] :#patient [is not open]]]
[203 [[Can't] :#agent [— file] :#patient [is not open in that mode]]]
[204 [[ill-formed list read from file] :#patient]]
[205 [[Can't open file] :#patient [— does file exist?]]]
```

5.3.3 PARSE.TRAP

```
[1 [:#advice [
I expected a ) to follow.]]]
[2 [:#advice [
I expected a } to follow.]]]
[3 [:#advice [
I expected a , to follow.]]]
[4 [[Bad reference to infix]]]
[5 [[You haven't told me how to] last :trapobject]]
[6 [[currently undocumented???? WDEX]]]
[7 [[extra right bracket on list]]]
[8 [[missing right bracket on list]]]
[9 [[The last procedure needs more inputs]]]
[10 [:trapobject [found — expression expected]]]
[11 [[I understand up through] butlast :trapobject [—] last :trapobject [is extra]]]
[12 [last last :trapobject [can't be part of a number] ]]
```

5.3.4 TRACE.TRAP

```
[400 [[traced on variable read]]]
[401 [[traced on variable write]]]
[402 [[traced on procedure call]]]
[403 [[traced on stepped procedure line]]]
[404 [[trace on procedure finish]]]
```